ЛЕГКОЕ ПРОГРАММИРОВАНИЕ

PYTHON ДЛЯ ДЕТЕЙ

САМОУЧИТЕЛЬ ПО ПРОГРАММИРОВАНИЮ

ДЖЕЙСОН БРИГГС





By Jason R. Briggs

PYTHON FOR KIDS

A PLAYFUL INTRODUCTION TO PROGRAMMING



San Francisco

Джейсон Бриггс

PYTHON ДЛЯ ДЕТЕЙ

САМОУЧИТЕЛЬ ПО ПРОГРАММИРОВАНИЮ

Москва
«Манн, Иванов и Фербер»
2017

УДК 087.5:004.43
ББК 76.1,62:32.973.412
Б87

Перевод с английского Станислава Ломакина

Издано с разрешения *No Starch Press, Inc., a California Corporation*

На русском языке публикуется впервые

Возрастная маркировка в соответствии
с Федеральным законом № 436-ФЗ: 0+

Бриггс, Джейсон

Б87 Python для детей. Самоучитель по программированию / Джейсон Бриггс ; пер. с англ. Станислава Ломакина ; [науч. ред. Д. Абрамова]. — М. : Манн, Иванов и Фербер, 2017. — 320 с.

ISBN 978-5-00100-616-9

Эта книга позволит вам погрузиться в программирование и с легкостью освоить Python. Вы сможете написать несколько настоящих игр. На каждом шагу вы будете видеть результаты своих трудов — в виде работающей программы, а с понятными инструкциями и примерами с забавными иллюстрациями обучение будет только приятным. Книга для детей от 10 лет.

УДК 087.5:004.43
ББК 76.1,62:32.973.412

ISBN 978-5-00100-616-9

ОГЛАВЛЕНИЕ

ЧАСТЬ III
ПИШЕМ ИГРУ «ЧЕЛОВЕЧЕК СПЕШИТ К ВЫХОДУ»

ВВЕДЕНИЕ

Зачем изучать программирование?

Программирование развивает креативность, логическое мышление, а также навыки поиска и устранения ошибок. Программист может создавать что-то из ничего, пользуясь логикой для составления понятных компьютеру программных конструкций, а если что-то пойдет не так, он отыщет ошибку и исправит проблему. Писать программы — занятие увлекательное и временами непростое, однако полученный опыт пригодится и в школе, и дома (даже если ваша профессия не будет связана с компьютерами).

Ну и, по меньшей мере, программирование — отличный способ скоротать время, когда за окном непогода.

Почему именно Python?

Python — простой в изучении язык программирования, он особенно хорош для начинающих. В отличие от многих других языков, Python-код легко читается, а интерактивная оболочка позволяет вводить программы и сразу же получать результат. Помимо простой структуры языка и интерактивной оболочки, в Python есть инструменты, заметно ускоряющие обучение и позволяющие создавать несложные анимации для своих видеоигр. Один из таких инструментов — специально созданный для обучения модуль `turtle`, который имитирует «черепашью графику» (в 1960-х годах она использовалась в языке Logo). Другой инструмент — модуль `tkinter` для работы с графической библиотекой `Tk`, позволяющей создавать программы с продвинутой графикой и анимацией.

Как изучать программирование?

Как правило, если вы встречаетесь с чем-то впервые, лучше начинать с основ, поэтому читайте книгу с самого начала, поборов искушение сразу перепрыгнуть в середину или конец. Никто не в силах сыграть симфонию, впервые взяв в руки инструмент. Начинающие пилоты не поднимаются в небо, не изучив приборы управления. Гимнасты не могут (как правило) сделать сальто назад с первой попытки. Если вы перейдете к последним главам раньше времени, вы не только плохо усвоите базовые понятия, но и сами эти главы покажутся вам куда сложнее, чем они есть на самом деле.

Читая книгу, запускайте каждый из примеров кода. В конце большинства глав есть упражнения, которые помогут укрепить знания. Если

что-то покажется вам непонятным или чересчур сложным, советую действовать так:

1. Разбейте задачу на составные части. Постарайтесь сперва понять, что делает небольшой фрагмент кода (фокусируйтесь на кусочках программы, не пытайтесь с ходу разобраться, как она устроена целиком).

2. Если это не помогает, иногда проблему лучше всего отложить, чтобы вернуться к ней на другой день. Этот способ хорош для многих жизненных ситуаций, и особенно при изучении программирования.

Для кого эта книга

Эта книга — для всех, кто интересуется программированием, будь это ребенок или взрослый, которому программирование в новинку. Если вы хотите не просто пользоваться чужими разработками, а создавать свое, «Python для детей» — хороший способ приступить к делу.

Изучив основы программирования, вы узнаете, как создавать собственные игры. Вам предстоит разработать две игры, научившись определять столкновения, использовать события и применять разные способы анимации.

Большинство примеров в этой книге рассчитаны на программирование в среде IDLE, которая идет в комплекте с Python. IDLE поддерживает подсветку синтаксиса, копирование и вставку текста, а также возможность сохранения и загрузки вашего кода. То есть IDLE одновременно и интерактивная среда для экспериментов, и что-то вроде текстового редактора. Хоть для запуска примеров достаточно стандартной консоли и обычного редактора текстов, подсветка синтаксиса и дружественный интерфейс IDLE облегчат вашу задачу, поэтому мы обязательно разберемся, как настроить и использовать эту среду.

Что вас ждет?

Вот краткое описание материала каждой из глав.
Глава 1 — введение в программирование и инструкции по установке Python.
Глава 2 — знакомство с простыми вычислениями и с переменными.
Глава 3 — описание некоторых основных типов данных (таких как строки, списки, кортежи).
Глава 4 — знакомство с модулем `turtle`. От основ программирования мы перейдем к перемещению черепашки (она похожа на стрелочку) по экрану.
Глава 5 — описание логических условий и конструкции `if`.

Глава 6 — циклы `for` и `while`.
Глава 7 — введение в создание и использование функций.
Глава 8 — введение в классы и объекты. На этом этапе мы освоим достаточно базовых возможностей языка, чтобы использовать приемы программирования игр.
Глава 9 — обзор большинства встроенных функций Python.
Глава 10 — обзор нескольких модулей, которые идут в комплекте с Python.
Глава 11 — снова о модуле `turtle` и рисовании более сложных фигур.
Глава 12 — модуль `tkinter` и создание продвинутой графики.
Главы 13 и 14 — пишем нашу первую игру, «Прыг-скок!», используя знания, полученные в предыдущих главах.
Главы от 15 до 18 — создаем вторую игру, «Человечек спешит к выходу». При вводе кода из глав, посвященных играм, вы можете допустить ошибки. Если найти их самостоятельно не получится, скачайте код игры с сайта этой книги (*python-for-kids.com/* или *mann-ivanov-ferber.ru*) и сравните с ним вашу программу.
Послесловие — краткий обзор модуля PyGame и некоторых других популярных языков программирования.
Приложение — подробное описание ключевых слов Python.
Глоссарий — определения терминов из области программирования, которые встречаются в данной книге.

Повеселитесь!

Изучая эту книгу, помните, что программирование может быть очень увлекательным. Воспринимайте его не как работу, а как способ создания веселых игр и программ, которыми можно поделиться с другими людьми. Изучение программирования отлично тренирует ум, и результаты могут быть впечатляющими. Но главное — что бы вы ни делали, не забывайте веселиться!

ЧАСТЬ I

Учимся программировать

1

НЕ ВСЕ ЗМЕИ ПРЕСМЫКАЮТСЯ

Компьютерная программа — это набор инструкций, следуя которым компьютер выполняет различные действия. Программу не найти среди деталей компьютера: проводов, микросхем, карт памяти, жестких дисков и тому подобного. Ее невозможно увидеть, однако выполняется она с помощью аппаратуры. Компьютерная программа (или просто *программа*) состоит из последовательности команд, указывающих оборудованию, что и как делать. Совокупность работающих на компьютере программ называют *программным обеспечением*.

Практически любое из электронных устройств, которыми мы пользуемся, не будет работать или станет гораздо менее полезным, если лишить его программного обеспечения. Программы управляют не только компьютерами, но и мобильными телефонами, игровыми приставками, автомобильными GPS-навигаторами. Среди не столь очевидных примеров — жидкокристаллические телевизоры, DVD-плееры, микроволновые печи и некоторые модели холодильников. Даже двигатели автомобилей, светофоры и уличные фонари, электронные рекламные панели и лифты в наши дни работают благодаря программам.

Программы чем-то похожи на мысли. Если бы у нас не было мыслей, мы, наверное, сидели бы на полу, ничего не делая. Мысль встать с пола — это инструкция, или команда, которая говорит нашему телу, что нужно подняться. Так же и программы говорят компьютеру, как ему действовать.

Научившись программировать, вы сможете делать множество полезных вещей. Вряд ли вы будете создавать программы для автомобилей, светофоров или холодильников (во всяком случае, это требует специальной подготовки), однако вы сможете разрабатывать веб-страницы,

видеоигры и даже писать программы, помогающие делать домашние задания.

Немного о языке

Как и люди, компьютеры «говорят» на разных языках, только языки эти — компьютерные. *Компьютерный язык* служит для того, чтобы переговариваться с компьютером, используя команды, понятные и компьютеру, и человеку.

Некоторые языки программирования названы в честь людей (например, Ада и Паскаль), другие названия являются простыми акронимами, то есть аббревиатурой (к примеру, BASIC — от англ. Beginner's All-purpose Symbolic Instruction Code, универсальный код символических инструкций для начинающих), и уж совсем немногие языки названы в честь телевизионных шоу — как Python. О да, язык программирования Python (произносится «Пайтон», с ударением на первый слог, хотя имейте в виду, что в России многие называют язык просто «питон») получил свое имя благодаря телешоу «Летающий цирк Монти Пайтона», так что змея питон здесь вовсе ни при чем.

Python — *букв.* питон

! *«Летающий цирк Монти Пайтона» — британское комедийное телешоу, впервые вышедшее на экраны в 1970 году. Хотя съемки «летающего цирка» давно прекращены, у него множество поклонников по всему миру. Среди комедийных скетчей этого шоу есть, например, зарисовки «Министерство глупых походок», «Рыбошлепский танец» и «Сырная лавка» (в которой не продают сыр).*

Благодаря некоторым особенностям Python отлично подходит для новичков. Главное — на нем можно писать простые и эффективные программы, не тратя на это много времени. В Python используется меньше сложных специальных символов, чем в большинстве других языков, так что программы на нем легко читаются. (Однако не думайте, что в программах на Python нет особых символов, просто они используются реже, чем во многих других языках.)

Установка Python

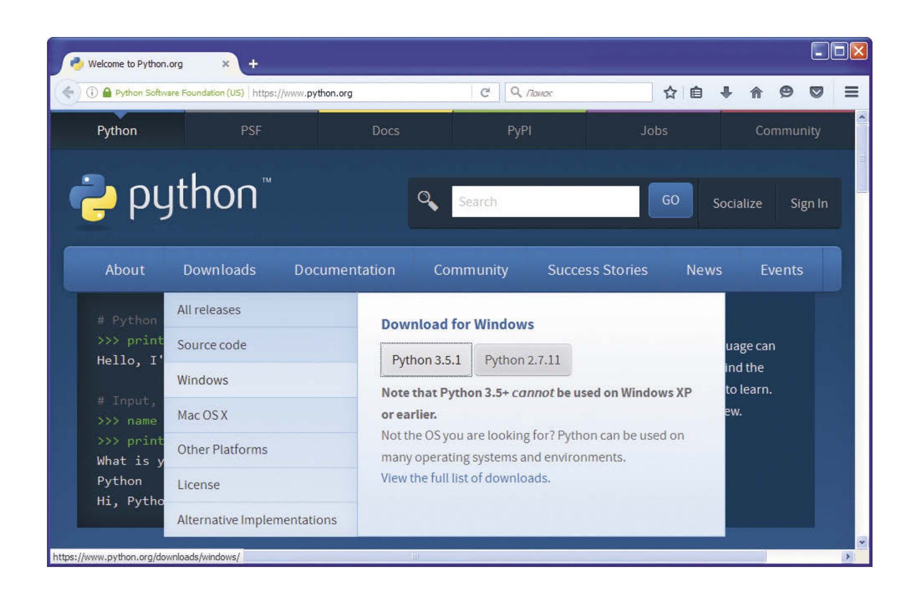
Установить Python на компьютер совсем несложно. Сейчас мы шаг за шагом разберем, как устанавливать его на системы Windows 7, Mac OS X и Ubuntu. Также мы создадим на рабочем столе ярлык для IDLE — среды разработки Python-программ. Если Python уже установлен на вашем компьютере, можете сразу переходить к разделу «Когда Python установлен» на стр. 20.

IDLE (от Integrated DeveLopment Environment) — интегрированная среда разработки

Установка Python в системе Windows 7

Чтобы установить Python в системе Microsoft Windows 7, откройте веб-браузер, введите адрес *http://www.python.org/* и скачайте последнюю версию программы-установщика Python 3 для Windows (для этого зайдите в меню *Downloads* и выберите *Windows*).

Download — скачать



! *Неважно, какую конкретно версию Python вы скачаете. Главное, чтобы ее номер начинался с цифры 3.*

После того как установщик скачается, дважды кликните мышкой по его значку и установите Python, следуя инструкциям программы:

1. Выберите **Install for All Users** и нажмите **Next**.
2. Не меняйте указанный адрес установки, но запомните его (например, *C:\Python31 или C:\Python32*). Нажмите **Next**.
3. Ничего не меняйте в разделе установщика *Customize Python*, просто нажмите **Next**.

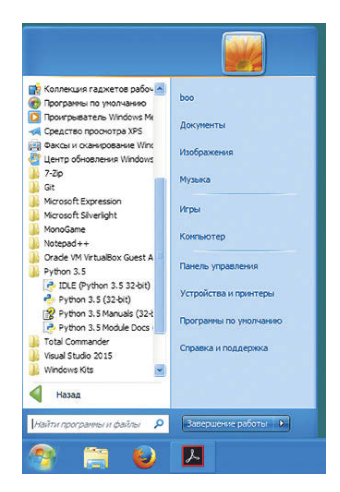
Install for All Users — установить для всех пользователей

Next — далее

Customize Python — настроить Python

После окончания установки в меню *Start (Пуск)* должен появиться раздел *Python 3*.

Start — пуск



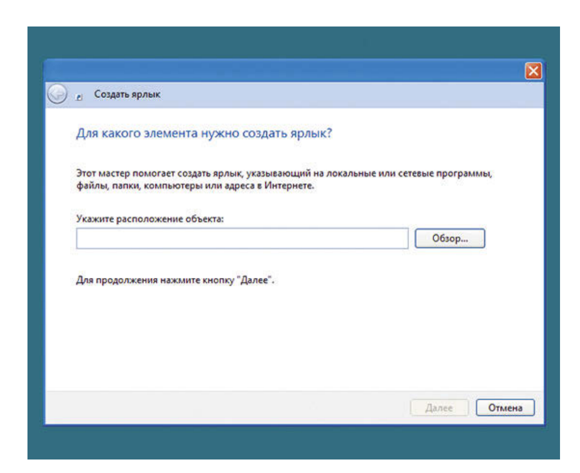
Теперь добавьте ярлык Python 3 на рабочий стол:

1. Кликните по рабочему столу правой кнопкой мышки и выберите из появившегося меню **New ▸ Shortcut** (**Создать ▸ Ярлык**).

2. Введите в поле с пометкой **Type the location of the item** (Укажите расположение объекта) следующую строку (каталог в начале этой строки должен соответствовать каталогу установки, который я просил вас запомнить):

```
c:\Python32\Lib\idlelib\idle.pyw -n
```

Диалоговое окно должно выглядеть так:



3. Нажмите **Next (Далее)**, чтобы перейти к следующему диалогу.

4. Укажите имя *IDLE* и нажмите **Finish (Готово)**, чтобы создать ярлык.

Теперь переходите к разделу «Когда Python установлен» на стр. 20 и начинайте знакомство с Python.

Установка Python в системе MAC OS X

Если у вас Mac, Python должен быть уже установлен в системе, однако скорее всего это одна из старых версий языка. Откройте веб-браузер, перейдите по адресу *http://www.python.org/getit/* и скачайте последнюю версию инсталлятора для Mac OS X.

Вам нужно выбрать инсталлятор в зависимости от вашей версии Mac OS X (чтобы узнать версию, кликните по значку с яблоком в верхнем меню и выберите пункт **About this Mac**).

После того как файл скачается, дважды клините по нему. Должно появиться окошко с содержимым файла.



В этом окне дважды кликните по значку *Python.mpkg* и следуйте инструкциям. Перед установкой система попросит вас ввести пароль администратора (не знаете пароль? Спросите у родителей).

Теперь добавьте на рабочий стол скрипт для запуска среды разработки IDLE:

1. Кликните по значку **Spotlight** — увеличительному стеклышку в правом верхнем углу экрана.

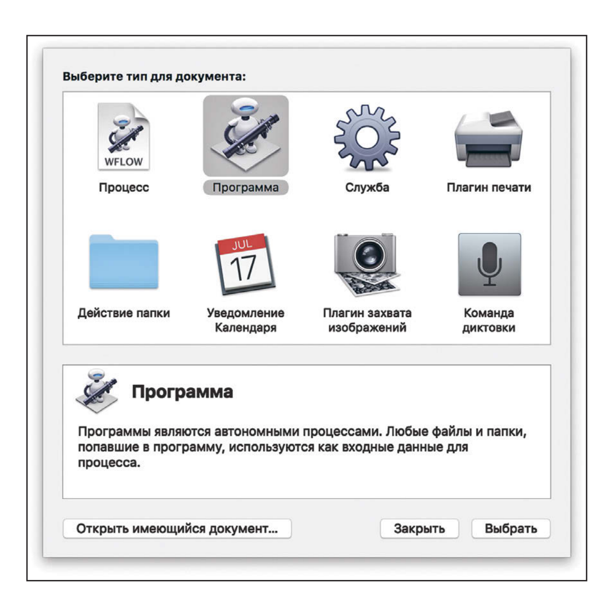
Spotlight — *букв.* прожектор

2. В появившемся поле введите *Automator*.

3. Кликните по приложению со значком в виде робота, когда оно появится в меню (либо в секции *Top Hit*, либо в секции *Applications*).

4. После того как Automator запустится, выберите шаблон **Application** (Программа).

Application — приложение

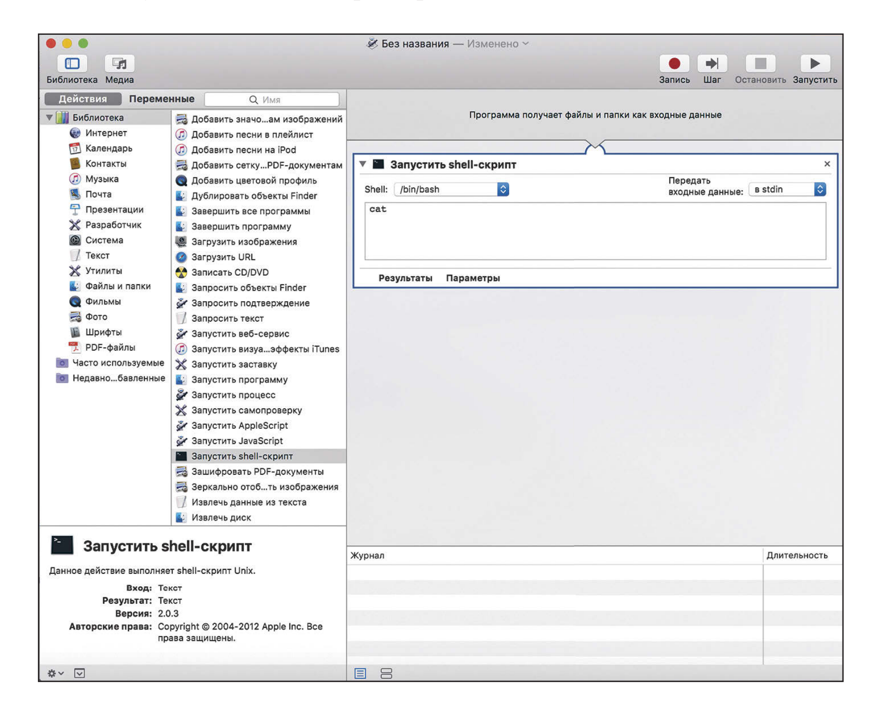


Choose — выбрать

5. Нажмите кнопку **Choose** (Выбрать).

Run — запустить
Shell Script — скрипт оболочки

6. Отыщите в списке действий пункт **Run Shell Script** (Запустить shell-скрипт) и перетащите его на пустую панель справа. Результат будет выглядеть примерно так:



7. В поле ввода вы увидите слово *cat*. Замените его такой строкой:

```
open -a "/Applications/Python 3.2/IDLE.app" —args -n
```

8. Выберите **File ▸ Save** (Файл ▸ Сохранить) и укажите в качестве имени *IDLE*.

9. В диалоге *Where* выберите **Desktop** (Рабочий стол) и нажмите **Save** (Сохранить).

Теперь переходите к разделу «Когда Python установлен» (на стр. 20) и начинайте знакомство с Python.

Установка Python в системе Ubuntu

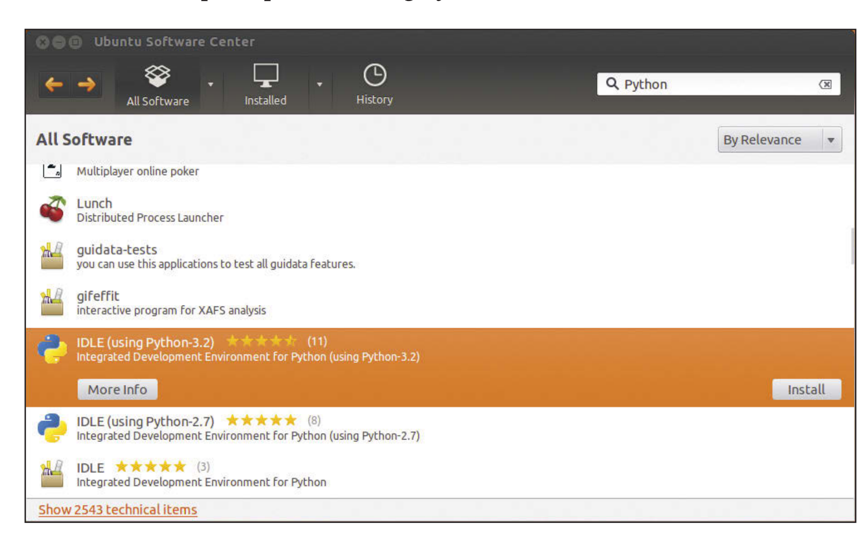
Python уже входит в Ubuntu Linux, но это может быть старая версия языка. Для установки Python 3 выполните следующие шаги:

1. В сайдбаре кликните по кнопке центра приложений Ubuntu (значок с оранжевой сумкой. Если вы его не видите, кликните по значку меню Dash и введите в строке поиска *Software*).

Software — программное обеспечение

2. Запустив центр приложений, введите *Python* в строке поиска (расположенной в правом верхнем углу окна).

3. В появившемся списке приложений выберите последнюю версию IDLE, например *IDLE (using Python 3.2)*:



4. Нажмите **Install**.

Authenticate — авторизация

5. Введите пароль администратора и нажмите **Authenticate** (не знаете пароль? Спросите у родителей).

! *В некоторых версиях Ubuntu в списке будет лишь строка Python (v3.2) — без слова IDLE. Это тоже подходит.*

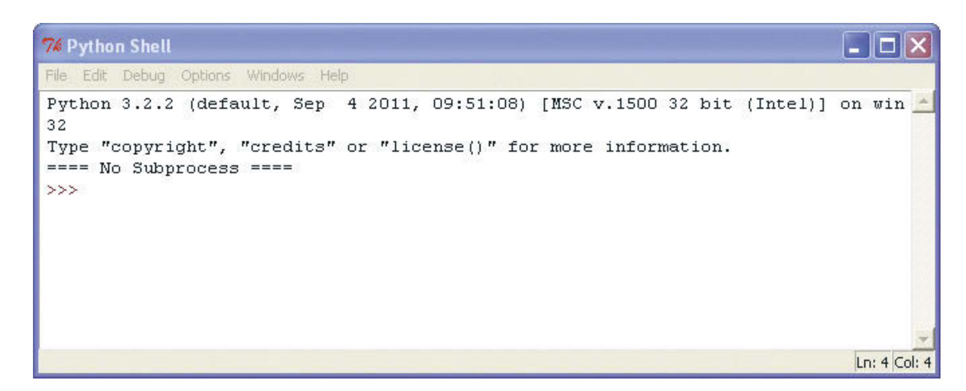
Итак, у вас установлена последняя версия Python. Давайте скорее посмотрим, что же это такое.



Когда Python установлен

Если вы пользуетесь Windows или Mac OS X, к этому моменту на вашем рабочем столе должен находиться значок с надписью *IDLE*. Если же вы используете Ubuntu, в меню *Applications* должен появиться раздел *Programming*, а в нем приложение *IDLE (using Python 3.2)* (или более поздняя версия).

Дважды кликните по значку или выберите приложение из меню. Должно появиться такое окно:



Это командная оболочка Python, которая входит в интегрированную среду разработки, а три знака «больше» (>>>) называются *приглашением*.

После приглашения можно вводить различные команды. Что ж, давайте приступим:

```
>>> print("Привет, мир")
```

Не забудьте про двойные кавычки (" "). Закончив вводить эту строку, нажмите клавишу **Enter**. Если вы ввели команду без ошибок, на экране должно появиться:

```
>>> print("Привет, мир")
Привет, мир
>>>
```

Приглашение возникнет снова. Это значит, что оболочка Python готова к выполнению дальнейших команд.

Поздравляю! Вы только что создали первую программу на языке Python! Слово `print` относится к разновидности команд, которые называются *функциями*, и эта конкретная функция выводит на экран все, что указано после нее в двойных кавычках. То есть вы дали компьютеру команду напечатать слова «Привет, мир» и эта команда понятна и вам, и компьютеру.

Print — печать



Сохранение Python-программ

От программ было бы мало толку, если бы их каждый раз приходилось писать заново. Конечно, если программа совсем короткая, это несложно. Однако большие программы могут состоять из миллионов строк кода. Чтобы распечатать весь код такой программы, например редактора документов, потребуется не меньше 100 000 листов бумаги. Представьте, каково нести такую груду листов домой!

К счастью, тексты программ можно сохранять на диск. Чтобы сохранить новую программу, запустите IDLE и выберите в меню **File ▸ New File**. Откроется пустое окно со словом *Untitled* в заголовке. Введите в этом новом окне такой код:

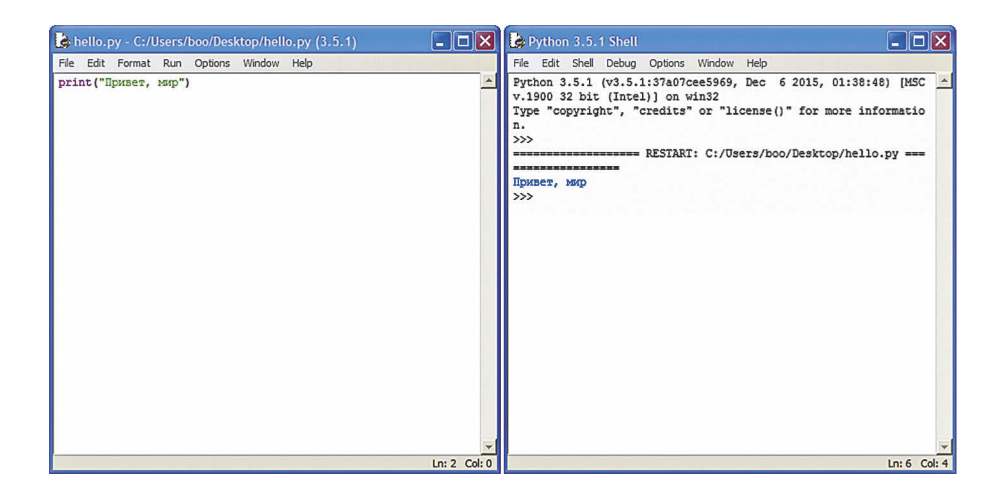
Untitled — без названия

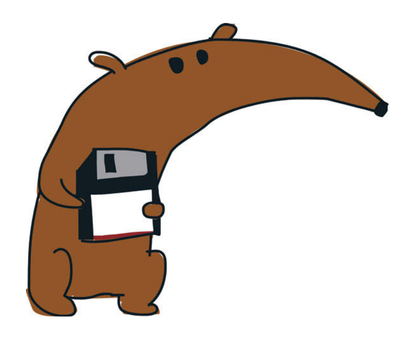
```
print("Привет, мир")
```

Теперь выберите в меню **File ▸ Save**. Введите в ответ на запрос имени файла *hello.py* и сохраните файл на рабочий стол. Теперь выберите **Run ▸ Run Module**. Сохраненная программа должна запуститься.

Run module — запустить модуль

Если вы закроете окно оболочки, оставив открытым окно с заголовком *hello.py,* и выберете из меню **Run ▸ Run Module**, окно оболочки появится снова и ваша программа запустится. Чтобы открыть оболочку Python без запуска программы, выберите **Run ▸ Python Shell**.





После сохранения и запуска этой программы вы обнаружите на рабочем столе новый значок с названием *hello.py*. Если дважды кликнуть по нему мышкой, на экране появится черное окно и тут же исчезнет. Что произошло?

Это было консольное окно Python (что-то вроде командной оболочки), где наша программа запустилась, напечатала слова «Привет, мир» и тут же завершила работу. Это окно практически невозможно рассмотреть, прежде чем оно закроется:



В IDLE для открытия окна оболочки, сохранения файла и запуска программы можно использовать не только команды меню, но и специальные сочетания клавиш:

- В системах Windows и Ubuntu используйте **CTRL-N** для создания нового окна, **CTRL-S** — для сохранения отредактированного файла и **F5** — для запуска программы.

- В системе Mac OS X используйте **⌘-N** для создания нового окна и **⌘-S** для сохранения файла, а для запуска программы нажмите клавишу **FN** и, удерживая ее, нажмите **F5**.

Что мы узнали

В этой главе мы создали программу «Привет, мир». По традиции с нее принято начинать изучение программирования. В следующей главе мы воспользуемся оболочкой Python для решения задач посложнее.

2

ВЫЧИСЛЕНИЯ И ПЕРЕМЕННЫЕ

Итак, вы установили Python и знаете, как запускать его командную оболочку, а значит, пора использовать его по назначению. Мы начнем с простых математических расчетов, а затем перейдем к важной части языка — переменным. *Переменные* — это удобный способ хранения данных в программе, и они пригодятся нам для решения самых разных задач.

Вычисления в Python

Если нужно перемножить два числа, к примеру узнать, сколько будет 8 × 3,57, мы обычно пользуемся калькулятором либо берем ручку и умножаем в столбик на листе бумаги. А что если использовать для подсчетов оболочку Python? Давайте попробуем.

Запустите оболочку, дважды кликнув по значку IDLE на рабочем столе, либо, если у вас система Ubuntu, кликнув по значку IDLE в меню **Applications**. Затем после значка >>> введите выражение и нажмите Enter:

```
>>> 8 * 3.57
28.56
```

Обратите внимание, что при записи числа 3,57 используется не запятая, а точка. Кроме того, в Python числа перемножаются с помощью звездочки (*), а не знака умножения (×).

Теперь рассмотрим более полезную задачу.

Представьте, что вы рыли яму и случайно нашли кошелек с 20 золотыми монетами. На следующий день вы тихонько залезли в подвал, где

стоит изобретение вашего дедушки — работающий на паровом ходу механизм для копирования предметов, и, на ваше счастье, в него удалось запихнуть все 20 монет. Раздался свист, потом щелчок, и устройство выдало еще 10 новеньких монеток.

Сколько монет вы накопите, если будете проделывать эту операцию каждый день в течение года? На бумаге эти расчеты выглядят примерно так:

$$10 \times 365 = 3650$$
$$20 + 3650 = 3670$$

Что ж, ничего сложного, осталось лишь выяснить, как посчитать то же в оболочке Python. Первым делом умножаем 10 монет на 365 дней, получится 3650. Затем добавим 20 монет, которые были изначально, и выйдет 3670.

```
>>> 10 * 365
3650
>>> 20 + 3650
3670
```

Но что если о вашем богатстве узнает пронырливая ворона? Предположим, она будет каждую неделю залетать в окно и красть по три монетки.

Сколько у вас будет монет через год? В оболочке Python эти расчеты будут выглядеть так:

```
>>> 3 * 52
156
>>> 3670 - 156
3514
```

Сперва умножаем 3 монеты на 52 недели в году, получаем 156. Затем вычитаем это значение из общего количества монет. Выходит, через год у вас останется 3514 монет.

Получилась очень простая программа. Изучая эту книгу дальше, вы узнаете, как писать более сложные и полезные программы.



Операторы в Python

В оболочке Python можно умножать, складывать, вычитать и делить числа, а также совершать некоторые другие операции, о которых мы узнаем позже. Символы, с помощью

которых выполняются математические действия в языке Python, называются *операторами*. Основные математические операторы перечислены в таблице 2.1.

Символ	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление

Таблица 2.1. Основные операторы в Python

Прямой слеш (/) обозначает деление, этот символ похож на линию между числителем и знаменателем дроби. Например, у вас 100 пиратов и 20 больших бочек, и вы хотите рассчитать, сколько пиратов можно спрятать в каждой бочке. Для этого следует разделить 100 пиратов на 20 бочек, введя в оболочке 100 / 20. И запомните — прямым слешем называют черту, верх которой наклонен вправо.

Порядок выполнения операций

Операции — это любые действия, которые совершаются с помощью операторов. Математические операции выполняются по очереди в зависимости от их приоритета (если не задать другую очередность с помощью скобок). Умножение и деление имеют более высокий приоритет, чем сложение и вычитание, и это значит, что они будут выполняться первыми. Иначе говоря, при вычислении математического выражения Python сначала умножит и разделит числа, а затем перейдет к сложению и вычитанию.

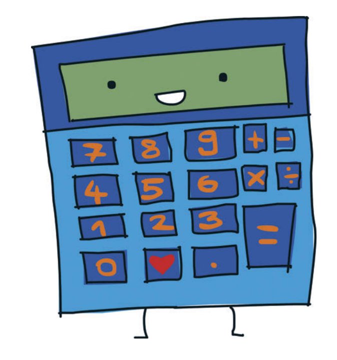
Например, в этом выражении сперва будут перемножены числа 30 и 20, а затем к их произведению будет прибавлено число 5.

```
>>> 5 + 30 * 20
605
```

По сути это выражение означает «умножить 30 на 20 и прибавить к результату 5». Получается 605. Однако мы можем изменить порядок операций, заключив первые два числа в скобки. Вот так:

```
>>> (5 + 30) * 20
700
```

В результате получилось 700, а не 605, поскольку Python выполняет операции в скобках прежде, чем операции вне скобок. Другими словами, это выражение означает «прибавить 5 к 30 и умножить результат на 20».



Скобки могут быть *вложенными*, то есть внутри скобок могут стоять еще одни скобки:

```
>>> ((5 + 30) * 20) / 10
70.0
```

В этом примере Python сперва вычислит выражение во внутренних скобках, затем во внешних и в самом конце выполнит стоящую за скобками операцию деления.

Иначе говоря, это выражение означает «прибавить 5 к 30, затем умножить результат на 20, потом разделить результат на 10». Вот что при этом происходит:

- сложение 5 и 30 дает 35;
- умножение 35 на 20 дает 700;
- деление 700 на 10 дает окончательный результат — 70.

Если бы мы не использовали скобки, результат вышел бы другим:

```
>>> 5 + 30 * 20 / 10
65.0
```

В этом случае сперва 30 умножается на 20 (получается 600), затем 600 делится на 10 (выходит 60) и, наконец, к 60 прибавляется 5, что дает в итоге 65.

! *Запомните, что умножение и деление всегда выполняются прежде, чем сложение и вычитание, если не менять порядок вычислений с помощью скобок.*

Переменные как ярлыки для данных

В программировании слово *переменная* обозначает именованное место для хранения данных, например чисел, текста, списков с числами или символами и так далее. Также переменную можно рассматривать как ярлык, которым помечены некие данные.

Например, чтобы создать переменную с именем fred, нужно указать имя, поставить знак «равно» (=) и ввести соответствующие данные. Давайте создадим переменную fred (Фред), указав, что ей соответствует значение 100 (однако из этого не следует, что другая переменная не может иметь такое же значение):

```
>>> fred = 100
```

Чтобы напечатать значение нашей переменной, введите в оболочке Python команду print и следом за ней — имя переменной в скобках. Вот так:

```
>>> print(fred)
100
```

Можно изменить значение переменной fred — сделать так, чтобы ей соответствовали другие данные. Например, вот как заменить значение fred числом 200:

```
>>> fred = 200
>>> print(fred)
200
```

В первой строке говорится, что переменной fred теперь соответствует число 200. Во второй строке мы запрашиваем значение fred, чтобы убедиться, что оно поменялось. Последней строкой Python печатает ответ.

Можно использовать несколько переменных для одного и того же значения:

```
>>> fred = 200
>>> john = fred
>>> print(john)
200
```

В этом примере знак «равно» между именами john (Джон) и fred говорит о том, что переменной john соответствует значение переменной fred.

Конечно, fred — не самое удачное имя переменной, поскольку оно не поясняет, для чего эта переменная используется. Лучше назовем переменную не fred, а, допустим, number_of_coins (количество монет):

```
>>> number_of_coins = 200
>>> print(number_of_coins)
200
```

Number of coins — количество монет

Теперь понятно, что речь идет о двухстах монетах.

Имена переменных могут состоять из латинских букв, цифр и знака подчеркивания (_), однако начинаться с цифры они не могут. В остальном допустимо использовать любые имена, которые могут состоять как из отдельных букв (например, a), так и из целых предложений (пробелы в именах недопустимы, но слова можно разделять знаками подчеркивания). Для небольших программ часто удобны короткие имена, но в целом желательно, чтобы имя переменной отражало смысл, который вы вкладываете в ее использование.

Теперь вы знаете, как создавать переменные. Давайте посмотрим, что с ними можно делать.

Использование переменных

Помните, как мы вычисляли, сколько монет накопится за год, если каждый день создавать новые монеты с помощью изобретения вашего дедушки? Итак, вот на чем мы остановились:

```
>>> 20 + 10 * 365
3670
>>> 3 * 52
156
>>> 3670 - 156
3514
```

Все это можно записать одной строкой кода:

```
>>> 20 + 10 * 365 - 3 * 52
3514
```

А что если заменить в этом выражении числа переменными? Введите:

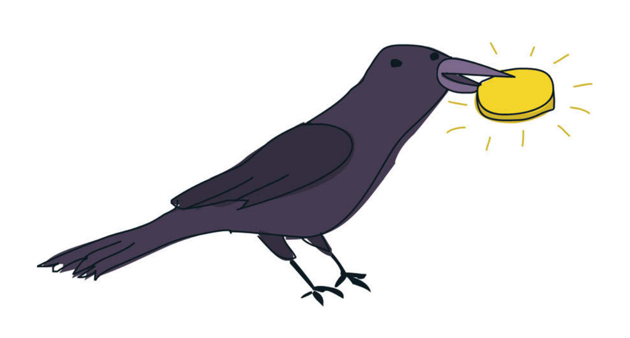
```
>>> found_coins = 20
>>> magic_coins = 10
>>> stolen_coins = 3
```

Found coins — найденные монеты
Magic coins — волшебные монеты
Stolen coins — украденные монеты

Мы создали три переменные: found_coins, magic_coins и stolen_coins.

Теперь можно записать наше выражение так:

```
>>> found_coins + magic_coins * 365 — stolen_coins * 52
3514
```



Как видите, результат остался прежним. А зачем все это? Ради особой магии переменных. Представьте, что вы поставили у окошка пугало и поэтому осторожная ворона крадет по две, а не по три монеты. Если использовать в расчетах переменную, достаточно поменять ее значение на другое число, чтобы оно использовалось везде, где эта переменная фигурирует. То есть мы можем поменять значение `stolen_coins`, введя:

```
>>> stolen_coins = 2
```

А затем скопировать и снова вставить наше выражение, чтобы пересчитать ответ. Вот так:

1. Выделите текст, который нужно скопировать. Для этого кликните по нему и, не отпуская кнопку мышки, перетащите область выделения от начала строки до ее конца, как показано на рисунке:

```
*Python Shell*
File  Edit  Debug  Options  Windows  Help
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> found_coins = 20
>>> magic_coins = 10
>>> stolen_coins = 3
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3514
>>> stolen_coins = 2
                                                            Ln: 7 Col: 55
```

2. Нажав и удерживая клавишу **Ctrl** (или, если вы используете Mac OS, клавишу ⌘), нажмите **C**, чтобы скопировать выделенный текст (далее я буду называть это действие **Ctrl-C**).

3. Кликните по строке с приглашением (следующей после `stolen_coins = 2`).

4. Нажав и удерживая **Ctrl**, нажмите **V**, чтобы вставить скопированный ранее текст (далее я буду называть это действие **Ctrl-V**).

5. Нажмите **Enter**, чтобы заново вычислить результат:

```
Python Shell
File Edit Debug Options Windows Help
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> found_coins = 20
>>> magic_coins = 10
>>> stolen_coins = 3
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3514
>>> stolen_coins = 2
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3566
>>> |
Ln: 12 Col: 4
```

Не правда ли, так гораздо проще, чем заново вводить все выражение? Можете поменять значения других переменных, а потом опять скопировать (**Ctrl-C**) и вставить (**Ctrl-V**) выражение, чтобы увидеть результат изменений. Предположим, вы обнаружили, что если стукнуть по дедушкиному изобретению, из него вылетает на 3 монеты больше. Поступая так каждый раз, через год вы получите 4661 монету:

```
>>> magic_coins = 13
>>> found_coins + magic_coins * 365 — stolen_coins * 52
4661
```

Конечно, в таком простом выражении от переменных *не очень* много толку. До настоящей пользы мы еще не дошли. Пока просто запомните, что переменные — это способ присваивать именам значения для их дальнейшего использования.

Что мы узнали

В этой главе мы разобрались, как составлять простые выражения с помощью операторов языка Python, использовать скобки для изменения порядка операций (очередности, в которой Python вычисляет части выражений), научились создавать переменные, присваивать им значения и использовать в расчетах.

3

СТРОКИ, СПИСКИ, КОРТЕЖИ И СЛОВАРИ

В предыдущей главе мы выполняли простые расчеты, а также познакомились с переменными. В этой главе мы научимся работать с еще несколькими конструкциями языка Python: строками, списками, кортежами и словарями. Строки пригодятся для вывода текста (например, сообщений «Старт!» или «Игра окончена» в компьютерной игре), а в списках, кортежах и словарях можно хранить наборы значений.

Строки

Фрагменты текста в программировании обычно называют *строками*. Можно сказать, что строка — это последовательность символов. Из всех букв, пробелов, цифр и других печатных знаков в этой книге можно составить строку, и из вашего имени или адреса тоже. По сути, первая программа на Python, которую мы создали в главе 1, уже включала в себя строку «Привет, мир».



Текст в кавычках нужно вводить без переносов. Здесь его пришлось перенести, поскольку он не вместился в ширину страницы, и это показано значком ↵. Встретив такой значок, вводите текст без переноса.

Создание строк

Чтобы создать строку, нужно ввести текст в кавычках — так Python отличает строки от чисел и других типов данных. Например, возьмем переменную `fred` из главы 2 и присвоим ей строковое значение:

```
fred = "Почему у горилл большие ноздри? Потому что у них ↵
толстые пальцы!"
```

Теперь напечатаем значение `fred`, воспользовавшись командой `print(fred)`:

```
>>> print(fred)
Почему у горилл большие ноздри? Потому что у них толстые пальцы!
```

Строку можно записать и в одинарных кавычках:

```
>>> fred = 'Что это: розовое и пушистое? Розовый пушистик!'
>>> print(fred)
Что это: розовое и пушистое? Розовый пушистик!
```

Однако если вы попытаетесь перенести текст, который начинается с одинарной (') или двойной (") кавычки на новую строку или поставить в начале текста кавычку одного типа, а в конце — другого, Python выдаст сообщение об ошибке. Например, введите:

```
>>> fred = "Что едят на полдник динозавры?
```

И вот что получится:

```
SyntaxError: EOL while scanning string literal
```

Python выдал сообщение о синтаксической ошибке, потому что, вопреки правилам, мы не завершили строку одинарной или двойной кавычкой.

Синтаксическая ошибка — это неверное расположение слов в предложении или — в нашем случае — слов и символов в программе. Сообщение `SyntaxError` означает, что вы ввели данные не в том порядке, который ожидает Python, или не ввели те данные, которые он от вас ждал. Здесь Python, дойдя до конца строки, не обнаружил закрывающую кавычку и выдал ошибку.

Syntax Error — синтаксическая ошибка

Если нужно ввести текст, занимающий несколько строк, поставьте в начале и в конце три одинарные кавычки, а когда понадобится сделать перенос, нажимайте **Enter**. Вот так:

```
>>> fred = '''Что едят на полдник динозавры?
ТиРекс-кекс!'''
```

Теперь напечатаем значение переменной fred:

```
>>> print(fred)
Что едят на полдник динозавры?
ТиРекс-кекс!
```

Проблемы со строками

Теперь посмотрите на следующую строку. Попытка ее ввести приведет к сообщению об ошибке:

Silly string — глупая строка
Invalid syntax — недопустимый синтаксис

```
>>> silly_string = '"Тут что-то не так, не будь я д'Артаньян" — ↵
подумал он.'
SyntaxError: invalid syntax
```

Мы попытались создать (и присвоить переменной silly_string) строку в одинарных кавычках, в которой есть двойные кавычки и еще одна одинарная в слове "д'Артаньян". Ну и беспорядок!

Не забывайте, что Python не обладает человеческим разумом, поэтому видит только строку, за которой следуют лишние символы. Когда Python встречает кавычку (одинарную или двойную) в начале строки, он считает, что такая же кавычка должна стоять и в конце. Поскольку в этом примере строка начинается с одинарной кавычки, Python воспринимает одинарную кавычку после буквы «д» как конец строки. Напечатав сообщение об ошибке, Python подсвечивает проблемное место в коде (это слово «Артаньян» после закрывающей кавычки):

```
Python 3.5.1 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> silly_string = '"Тут что-то не так, не будь я д'Артаньян" - подумал он.'
SyntaxError: invalid syntax
>>> |
                                                              Ln: 5  Col: 4
```

Последней строкой IDLE сообщает тип ошибки. В данном случае это ошибка синтаксическая.

Если вместо одинарных кавычек заключить нашу строку в двойные, ошибка все равно возникнет, хоть и в другом месте:

```
>>> silly_string = ""Тут что-то не так, не будь я д'Артаньян" — ↵
подумал он."
SyntaxError: invalid syntax
```

Теперь Python считает, что это строка в двойных кавычках, которая началась и тут же закончилась, и стоящее после закрывающей кавычки слово «Тут» вызывает ошибку:

```
Python 3.5.1 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> silly_string = ""Тут что-то не так, не будь я д'Артаньян" - подумал он."
SyntaxError: invalid syntax
>>> |
                                                              Ln: 5  Col: 4
```

С точки зрения Python, после закрывающей кавычки символов быть не должно — он просто не знает, что с ними делать.

Чтобы решить эту проблему, можно поставить три одинарные кавычки в начале и в конце строки (мы так уже делали, вводя строку с переносами). Это позволит использовать в тексте двойные и одинарные кавычки без риска ошибок. Тогда можно поместить внутрь строки любую комбинацию кавычек, кроме трех одинарных подряд. Вот корректный вариант строки:

```
silly_string = '''"Тут что-то не так, ↵
не будь я д'Артаньян", — подумал он.'''
```



Если вы очень хотите записать эту строку в двойных или одинарных кавычках, можно ставить перед каждым символом кавычек в тексте обратный слеш (\). Такой способ называется *экранированием*. Этим мы как бы говорим: да, внутри этой строки есть кавычки, но их не нужно воспринимать как закрывающие.

Экранирование усложняет восприятие строк, поэтому желательно пользоваться тремя одинарными кавычками. Однако вы можете встретить фрагменты чужого кода с экранированием, так что стоит понимать, к чему там используются все эти обратные слеши.

Вот примеры экранирования:

Single quote — одинарные кавычки
Double quote — двойные кавычки

```
❶ >>> single_quote_str = '"Тут что-то не так, не будь я д\'Артаньян", — ↵
подумал он.'
❷ >>> double_quote_str = "\"Тут что-то не так, не будь ↵
я д'Артаньян\", — подумал он."
>>> print(single_quote_str)
"Тут что-то не так, не будь я д'Артаньян", — подумал он.
>>> print(double_quote_str)
"Тут что-то не так, не будь я д'Артаньян", — подумал он.
```

В строке ❶ мы создали строку текста в одинарных кавычках, поставив перед одинарной кавычкой внутри нее обратный слеш. В строке ❷ мы создали строку текста в двойных кавычках, экранировав обратным слешем двойные кавычки внутри нее. Затем мы вывели только что созданные переменные на экран. Обратите внимание, что обратный слеш при этом напечатан не был.

Переменные внутри строк

My score — мой результат

Можно печатать строки, содержащие значения переменных. Для этого используются метки `%s` в тех местах, где должны быть значения (это называется *подстановкой*, или встраиванием значений в строку). Например, в переменной `myscore` хранится счет игры, и вы хотите, чтобы он отображался в сообщении «Мой счет:________очков». Тогда вместо числа используйте в сообщении метку `%s`, а при печати укажите нужное значение. Вот так:

```
>>> myscore = 1000
>>> message = 'Мой счет: %s очков'
>>> print(message % myscore)
Мой счет: 1000 очков
```

Message — сообщение

Мы создали переменную `myscore` со значением 1000 и переменную `message` со строкой «Мой счет: %s очков», где `%s` — метка позиции, в которой нужно отобразить значение. В следующей строке программы мы

используем команду print(message), указав с помощью знака %, что метку %s нужно заменить значением переменной myscore. В результате на экране появляется сообщение «Мой счет: 1000 очков». Для подстановки значения использовать переменную необязательно, то же самое получится, если дать команду print(message % 1000).

Печатая строку с меткой %s, можно каждый раз подставлять в нее значения разных переменных, как в этом примере:

```
>>> joke_text = '%s: приспособление для поиска мебели в темноте'
>>> bodypart1 = 'Коленка'
>>> bodypart2 = 'Лодыжка'
>>> print(joke_text % bodypart1)
Коленка: приспособление для поиска мебели в темноте
>>> print(joke_text % bodypart2)
Лодыжка: приспособление для поиска мебели в темноте
```

Мы создали три переменные. Первая, joke_text, содержит строку с меткой %s. Значения двух других переменных — bodypart1 и bodypart2 — строки с названиями частей тела. Теперь можно печатать joke_text, подставляя с помощью оператора % вместо %s разные переменные: bodypart1 или bodypart2. В том и другом случае на экране появятся разные сообщения.

Joke text — шуточный текст
Bodypart — часть тела

Можно использовать и несколько меток в одной строке. Вот так:

```
>>> nums = 'Что сказало число %s числу %s? Славный поясок!'
>>> print(nums % (0, 8))
Что сказало число 0 числу 8? Славный поясок!
```

Nums — от numbers — числа

Если меток несколько, указывайте значения для подстановки в скобках в том же порядке, что и в строке, как показано в этом примере.

Умножение строк

Что получится, если умножить 10 на 5? Разумеется, 50. А если умножить на 10 букву «*а*»? Вот что думает об этом Python:

```
>>> print(10 * 'a')
aaaaaaaaaa
```

Эта особенность может пригодиться для вывода строк с отступом в заданное число пробелов. Давайте напечатаем в оболочке Python такое письмо (выберите в меню **File ▸ New File**, и введите эту программу в новом окне):

```
spaces = ' ' * 25
print('%s Задний переулок 12' % spaces)
print('%s Трясогузочья пустошь' % spaces)
print('%s Западный Всхрапшир' % spaces)
print()
print()
print('Уважаемый Сэр,')
print()
print('Хочу сообщить вам, что кое-где на крыше уборной')
print('недостает кусков черепицы.')
print('Думаю, прошлой ночью их сдуло внезапным порывом ветра.')
print()
print('С почтением')
print('Малькольм Конфузли')
```

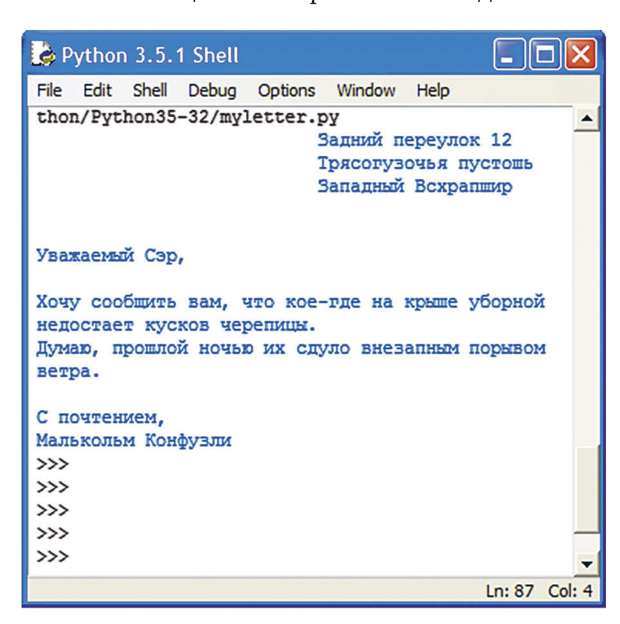
Введя этот код в новом окне оболочки, сохраните его под именем *myletter.py*, выбрав в меню **File ▸ Save As**. Теперь можете запустить программу, выбрав **Run ▸ Run Module** (как мы уже делали).



*Увидев перед фрагментом кода указание сохранить его под каким-то именем, знайте, что нужно выбрать в меню **File ▸ New File**, ввести код в появившемся окне и сохранить его так же, как в этом примере.*

Spaces — пробелы

Мы создали переменную `spaces`, присвоив ей результат умножения символа «пробел» на 25 (то есть строку из 25 пробелов). В следующих трех строках программы мы воспользовались этой переменной, чтобы напечатать блок текста со смещением вправо. Вот что должно получиться:



Помимо выравнивания блоков текста, умножение пригодится для заполнения экрана надоедливыми сообщениями. Запустите этот пример самостоятельно:

```
>>> print(1000 * 'слякоть')
```

Списки мощнее строк



Паучьи лапки, жабий палец, глаз тритона, крыло летучей мыши, жир слизня и перхоть змеи — довольно необычный список покупок (если только вы не колдун или ведьма), но мы воспользуемся им, чтобы показать разницу между строками и списками.

Можно представить этот список в виде строки, поместив ее в переменную `wizard_list`:

```
>>> wizard_list = 'Паучьи лапки, жабий палец, глаз тритона, крыло ↵
летучей мыши, жир слизня, перхоть змеи'
>>> print(wizard_list)
Паучьи лапки, жабий палец, глаз тритона, крыло летучей мыши, жир
слизня, перхоть змеи
```

Wizard list — волшебный список

Но лучше воспользоваться *списком* — специальным колдовским объектом языка Python. Вот как будут выглядеть наши ингредиенты в виде списка:

```
>>> wizard_list = ['паучьи лапки', 'жабий палец', 'глаз тритона', ↵
'крыло летучей мыши', 'жир слизня', 'перхоть змеи']
>>> print(wizard_list)
['паучьи лапки', 'жабий палец', 'глаз тритона', 'крыло летучей
мыши', 'жир слизня', 'перхоть змеи']
```

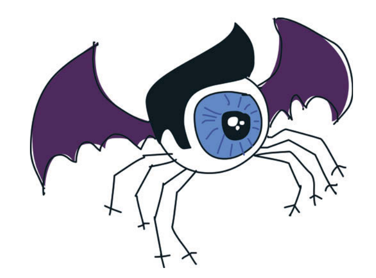
Чтобы создать список, понадобится ввести чуть больше символов, чем при создании строки, зато список позволяет обращаться к его элементам (в нашем случае, к покупкам) по отдельности. Например, мы можем напечатать третий элемент из `wizard_list`, указав в квадратных скобках его позицию (которую называют *индексом*), вот так:

```
>>> print(wizard_list[2])
глаз тритона
```

Что такое? Почему в скобках стоит 2, это же третий элемент в списке! Да, так и есть, однако отсчет элементов в списках начинается с нуля — то есть первому элементу соответствует индекс 0, второму 1, а третьему 2. Человеку это может показаться странным, однако для компьютеров такая нумерация естественна.

Также можно заменить элемент списка — это гораздо проще, чем заменить часть строки. Положим, нам требуется не глаз тритона, а язык улитки. Не проблема, давайте подправим список:

```
>>> wizard_list[2] = 'язык улитки'
>>> print(wizard_list)
['паучьи лапки', 'жабий палец', 'язык улитки', 'крыло летучей мыши', 'жир слизня', 'перхоть змеи']
```



Итак, мы заменили элемент с индексом 2 — прежде это был глаз тритона — языком улитки.

Также мы можем отобразить только часть элементов списка — это делается с помощью двоеточия (:) внутри квадратных скобок. Например, чтобы вывести на экран элементы с третьего до пятого (отличные ингредиенты для бутерброда), введите:

```
>>> print(wizard_list[2:5])
['язык улитки', 'крыло летучей мыши', 'жир слизня']
```

Запись [2:5] означает «показать элементы, начиная с индекса 2 и до индекса 5 (но не включая его)», иначе говоря, элементы 2, 3 и 4.

В списках можно хранить значения разных типов, например числа:

Some numbers — некоторые числа

```
>>> some_numbers = [1, 2, 5, 10, 20]
```

Или строки:

Some strings — некоторые строки

```
>>> some_strings = ['Нож', 'отточен', 'точен', 'очень']
```

Или числа и строки вперемежку:

```
>>> numbers_and_strings = [7, 'раз', 'отпей', 1, 'раз', 'отъешь']
>>> print(numbers_and_strings)
[7, 'раз', 'отпей', 1, 'раз', 'отъешь']
```

В списках могут даже храниться другие списки:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> strings = ['хватит', 'циферки', 'считать']
>>> mylist = [numbers, strings]
>>> print(mylist)
[[1, 2, 3, 4, 5], ['хватит', 'циферки', 'считать']]
```

Numbers — числа
Strings — строки
My list — мой список

Мы создали три переменные: numbers с пятью цифрами, strings с тремя строками и mylist, где хранятся списки numbers и strings. Причем в третьем списке (mylist) только два элемента, ведь он содержит два других списка, а не их отдельные элементы.

Добавление элементов в список

Для добавления в список новых элементов служит функция append. *Функция* — это фрагмент кода, который выполняет какую-то задачу. В данном случае append добавляет элемент к концу списка.

Append — добавить

Например, чтобы добавить в список колдовских покупок медвежий коготь, введем:

```
>>> wizard_list.append('медвежий коготь')
>>> print(wizard_list)
['паучьи лапки', 'жабий палец', 'язык улитки', 'крыло летучей
мыши', 'жир слизня', 'перхоть змеи', 'медвежий коготь']
```

Тем же способом можно и дальше добавлять в колдовской список ингредиенты:

```
>>> wizard_list.append('мандрагора')
>>> wizard_list.append('болиголов')
>>> wizard_list.append('болотный газ')
```

Теперь наш список выглядит так:

```
>>> print(wizard_list)
['паучьи лапки', 'жабий палец', 'язык улитки', 'крыло летучей мыши',
'жир слизня', 'перхоть змеи', 'медвежий коготь', 'мандрагора',
'болиголов', 'болотный газ']
```

Судя по всему, готовится нешуточная шалость!

Удаление элементов из списка

Del — от delete — удалить

Для удаления элементов из списка служит команда `del`. Например, чтобы удалить из списка колдовских ингредиентов шестой элемент (змеиную перхоть), введем:

```
>>> del wizard_list[5]
>>> print(wizard_list)
['паучьи лапки', 'жабий палец', 'язык улитки', 'крыло летучей мыши', 'жир слизня', 'медвежий коготь', 'мандрагора', 'болиголов', 'болотный газ']
```

! *Не забывайте, что позиции в списке отсчитываются с нуля, поэтому* `wizard_list[5]` *означает шестой элемент списка.*

А вот как можно удалить ингредиенты, которые мы добавили последними (мандрагору, болиголов и болотный газ):

```
>>> del wizard_list[8]
>>> del wizard_list[7]
>>> del wizard_list[6]
>>> print(wizard_list)
['паучьи лапки', 'жабий палец', 'язык улитки', 'крыло летучей мыши', 'жир слизня', 'медвежий коготь']
```

Списковая арифметика

Списки можно объединять, складывая их так же, как числа, с помощью знака «плюс» (+). Например, у нас есть два списка: `list1`, в котором хранятся числа от 1 до 4, и `list2`, где хранится несколько слов. Тогда мы можем сложить их, воспользовавшись командой `print` и знаком +. Вот так:

```
>>> list1 = [1, 2, 3, 4, 5]
>>> list2 = ['я', 'забрался', 'под', 'кровать']
>>> print(list1 + list2)
[1, 2, 3, 4, 5, 'я', 'забрался', 'под', 'кровать']
```

Результат сложения двух списков можно поместить в другую переменную:

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = ['я', 'мечтаю', 'о', 'пломбире']
>>> list3 = list1 + list2
>>> print(list3)
[1, 2, 3, 4, 'я', 'мечтаю', 'о', 'пломбире']
```

Также можно умножить список на число с помощью оператора (*). Например, умножим `list1` на 5:

```
>>> list1 = [1, 2]
>>> print(list1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Фактически это означает «повторить `list1` пять раз», поэтому в итоге получается 1, 2, 1, 2, 1, 2, 1, 2, 1, 2.

Но обратите внимание — деление и вычитание со списками не работают. При попытке сделать это вы получите ошибку, как в следующих примерах:

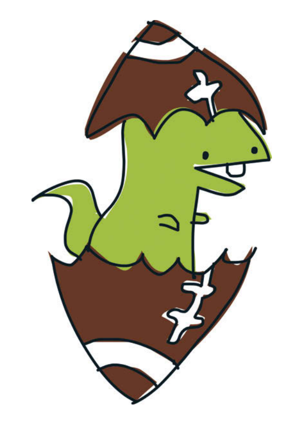
```
>>> list1 / 20
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
list1 / 20
TypeError: unsupported operand type(s) for /: 'list' and 'int'

>>> list1 - 20
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
list1 - 20
TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

Но почему? Дело в том, что объединение списков через + и повторение списков с помощью * — это очевидные действия, у которых есть аналоги в обычной жизни. Например, если я дам вам два списка покупок и попрошу их объединить, вы можете переписать все пункты этих списков на новый лист бумаги. Также, если я попрошу умножить этот список на 3, вы трижды перепишете на новый лист все пункты списка.

Ну а как вы будете список делить? Представьте, что вам нужно разделить список из шести чисел (от 1 до 6) на две части. Вот лишь три из множества вариантов:

```
[1, 2, 3]       [4, 5, 6]
[1]             [2, 3, 4, 5, 6]
[1, 2, 3, 4]    [5, 6]
```



Вы разделите список поровну, отделите первый пункт от остальных или возьмете две части наобум? Единственного правильного ответа здесь нет, поэтому и Python, если попросить его разделить список, не знает, что делать, и выдает ошибку.

Также не выйдет сложить список с несписковым значением. Например, вот что произойдет при попытке прибавить число 50 к list1:

```
>>> list1 + 50
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
list1 + 50
TypeError: can only concatenate list (not "int") to list
```

Почему возникла ошибка? Как именно прибавлять 50 к списку? Прибавить 50 к каждому его элементу? А если элементы не числовые? Может, имеется в виду, что 50 нужно добавить в начало списка как элемент?

В программировании команды должны быть однозначными, чтобы каждый раз, когда вы вводите команду, происходило одно и то же. Бестолковые компьютеры понимают только крайности — черное или белое. Дайте компьютеру задание с несколькими вариантами решений и получите в ответ кучу ошибок.

Кортежи

Кортеж похож на список, элементы которого записаны в круглых скобках, как в этом примере:

Fibs — *здесь* от fibonacci sequence — последовательность Фибоначчи

```
>>> fibs = (0, 1, 1, 2, 3)
>>> print(fibs[3])
2
```

Мы определили переменную fibs как набор чисел 0, 1, 1, 2 и 3. И точно так же, как со списками, мы напечатали элемент с индексом 3 с помощью команды print(fibs[3]).

Главное отличие кортежа от списка в том, что кортеж невозможно изменить после его создания. Например, если мы попытаемся поменять первое значение кортежа fibs на число 4 (таким же образом, каким меняли значения в списке wizard_list), мы получим сообщение об ошибке:

```
>>> fibs[0] = 4
Traceback (most recent call last):
```

```
File "<pyshell>", line 1, in <module>
fibs[0] = 4
TypeError: 'tuple' object does not support item assignment
```

Но в чем смысл использования кортежей, если есть списки? Главная причина такова: порой удобно использовать набор значений, который никогда не меняется. Создав кортеж с двумя элементами, можно не сомневаться, что в нем и дальше будут только эти два элемента.

Словари в Python — не для поиска слов

Словарями в Python называются наборы значений аналогично спискам и кортежам. Отличие состоит в том, что каждому элементу словаря соответствуют *ключ* и связанное с ним *значение*.

Например, у нас есть перечень людей и их любимых видов спорта. Можно поместить эту информацию в список, где следом за именем человека указан вид спорта. Вот так:

```
>>> favorite_sports = ['Ральф Уильямс, Футбол',
'Майкл Типпетт, Баскетбол',
'Эдвард Элгар, Бейсбол',
'Ребекка Кларк, Нетбол',
'Этель Смит, Бадминтон',
'Фрэнк Бридж, Регби']
```

Если я спрошу, какой у Ребекки Кларк любимый вид спорта, вы можете пробежать список глазами и обнаружить, что это нетбол. Но что если в списке 100 или больше людей?

Если мы сохраним те же данные в словаре, сделав имя человека ключом, а вид спорта значением, у нас получится следующий код:

```
>>> favorite_sports = {'Ральф Уильямс': 'Футбол',
'Майкл Типпетт': 'Баскетбол',
'Эдвард Элгар': 'Бейсбол',
'Ребекка Кларк': 'Нетбол',
'Этель Смит': 'Бадминтон',
'Фрэнк Бридж': 'Регби'}
```

Для разделения каждой пары «ключ–значение» мы использовали двоеточие, записав при этом ключ и значение в одинарных кавычках. Также обратите внимание, что элементы словаря заключены в фигурные (а не круглые или квадратные) скобки.

В результате получается словарь, где каждому ключу соответствует определенное значение, как показано в таблице 3.1.

Ключ	Значение
Ральф Уильямс	Футбол
Майкл Типпетт	Баскетбол
Эдвард Элгар	Бейсбол
Ребекка Кларк	Нетбол
Этель Смит	Бадминтон
Фрэнк Бридж	Регби

Таблица 3.1. Ключи и соответствующие им значения в словаре любимых видов спорта

Favorite sports — любимые виды спорта

Теперь, чтобы узнать любимый вид спорта Ребекки Кларк, нужно обратиться к словарю favorite_sports, использовав ее имя в качестве ключа:

```
>>> print(favorite_sports['Ребекка Кларк'])
Нетбол
```

Ответ: нетбол.

Чтобы удалить значение из словаря, тоже используется ключ. Например, удалим Этель Смит:

```
>>> del favorite_sports['Этель Смит']
>>> print(favorite_sports)
{'Ральф Уильямс': 'Футбол', 'Эдвард Элгар': 'Бейсбол', 'Фрэнк
Бридж': 'Регби', 'Майкл Типпетт': 'Баскетбол', 'Ребекка Кларк':
'Нетбол'}
```

Ключ нужен и для замены значения в словаре:

```
>>> favorite_sports['Ральф Уильямс'] = 'Хоккей на льду'
>>> print(favorite_sports)
{'Ральф Уильямс': 'Хоккей на льду', 'Эдвард Элгар': 'Бейсбол',
'Фрэнк Бридж': 'Регби', 'Майкл Типпетт': 'Баскетбол', 'Ребекка
Кларк': 'Нетбол'}
```

Здесь мы поменяли любимый вид спорта Ральфа Уильямса, указав вместо футбола хоккей на льду.

Как видите, работа со словарями напоминает работу со списками и кортежами, однако объединять словари с помощью оператора "+" нельзя. Попытавшись это сделать, вы получите сообщение об ошибке:

```
>>> favorite_sports = {'Ребекка Кларк': 'Нетбол',
'Майкл Типпетт': 'Баскетбол',
'Ральф Уильямс': 'Хоккей на льду',
'Эдвард Элгар': 'Бейсбол',
'Фрэнк Бридж': 'Регби'}
>>> favorite_colors = {'Малькольм Уорнер': 'Розовый горошек',
'Джеймс Бакстер': 'Оранжевые полоски',
'Сью Ли': 'Пурпурный орнамент'}
>>> favorite_sports + favorite_colors
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Python отказывается объединять словари, потому что не знает, как это делать.

Что мы узнали

В этой главе мы узнали, как хранить текст в строках и наборы значений в списках и кортежах. Можно менять элементы списков, объединять их, однако значения в кортеже изменить нельзя. Также мы научились пользоваться словарями для хранения значений, которым соответствуют ключи.

Упражнения

Вот несколько заданий для самостоятельного выполнения. Варианты их решений можно найти на сайте *http://python-for-kids.com/*, а также на страничке книги на сайте *http://mann-ivanov-ferber.ru*.

#1. Любимые вещи

Games — игры
Foods — блюда

Создайте список своих любимых развлечений и сохраните его в переменной `games`. Теперь создайте список любимых лакомств, сохранив его в переменной `foods`. Объедините два этих списка, сохранив результат в переменной `favorites`, и напечатайте значение этой переменной.

#2. Подсчет воинов

Есть 3 дома, на крыше каждого из которых прячутся по 25 ниндзя, и есть 2 туннеля, в каждом из которых скрывается по 40 самураев. Сколько всего воинов решили устроить заварушку? (Ответ можно найти, введя в оболочке Python арифметическое выражение.)

#3. Приветствие

Создайте две переменные: пусть одна хранит ваше имя, а другая фамилию. Теперь с помощью строки с метками `%s` напечатайте приветствие вроде такого: «Привет, Брандо Икетт!».

4

РИСОВАНИЕ С ПОМОЩЬЮ ЧЕРЕПАШКИ

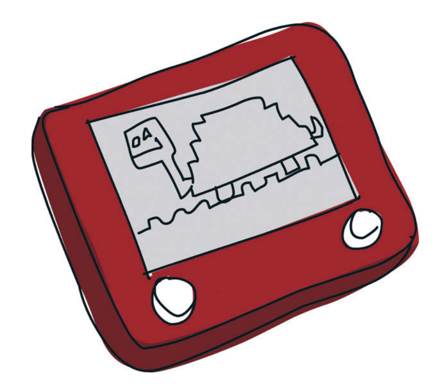
В мире Python *черепашкой* зовется объект, напоминающий обыкновенную черепаху, которая медленно ползает и носит свой дом на спине. Только в Python это небольшая черная стрелочка, которая медленно перемещается по экрану.

Черепашка хорошо подходит для изучения основ компьютерной графики, и в этой главе мы будем рисовать с ее помощью несложные контуры и линии.

Использование модуля черепашки

В Python *модулем* называется способ подключения полезного кода к другой программе, и в числе прочего модули обычно содержат функции, к которым можно обращаться. Подробнее мы поговорим об этом в главе 7, а пока скажу лишь, что в Python есть специальный модуль под названием `turtle`, которым мы и воспользуемся, чтобы изучить основы создания изображений на экране. С помощью модуля `turtle` можно программировать векторную графику, то есть составлять рисунки из линий, точек и кривых.

Turtle — черепаха



Давайте посмотрим, как работает `turtle`. Запустите оболочку Python, кликнув по значку на рабочем столе (или, если вы используете Ubuntu, выбрав в меню **Applications ▸ Programming ▸ IDLE**). Затем следует указать, что мы хотим использовать модуль `turtle`, и для этого его нужно *импортировать*. Вот так:

```
>>> import turtle
```

Импортирование модуля сообщает Python о том, что мы собираемся этот модуль использовать.



Если вы пользуетесь Ubuntu и, попытавшись импортировать модуль `turtle`*, получили ошибку, скорее всего, вам нужно установить пакет* `tkinter`*. Для этого откройте центр приложений Ubuntu и введите в строке поиска python-tk. В списке должен появиться пункт «Tkinter — Writing Tk Applications with Python». Кликните* ***Install****, чтобы установить этот пакет.*

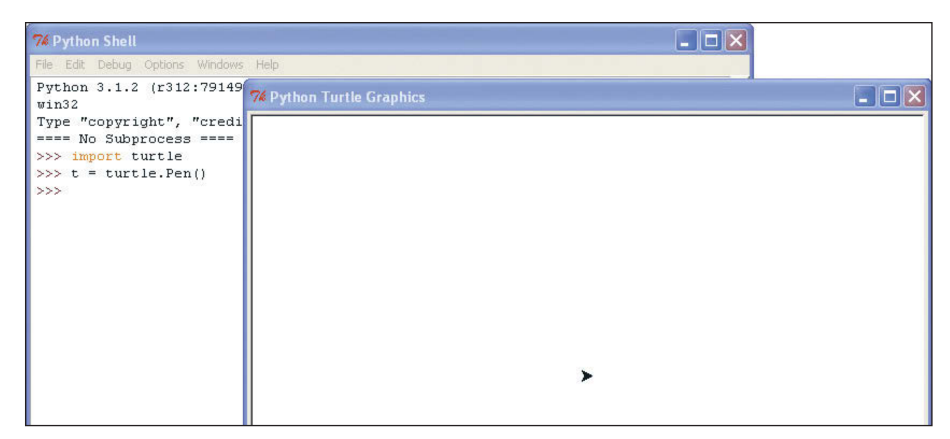
Создание холста

Итак, мы импортировали модуль `turtle`, и теперь нужно создать *холст* — чистое пространство для рисования, вроде холста у художников. Для этого вызовем функцию `Pen` из модуля `turtle`, и она автоматически создаст холст (о том, что такое «функция», мы поговорим позже). Введите в оболочке Python такую команду:

Pen — ручка

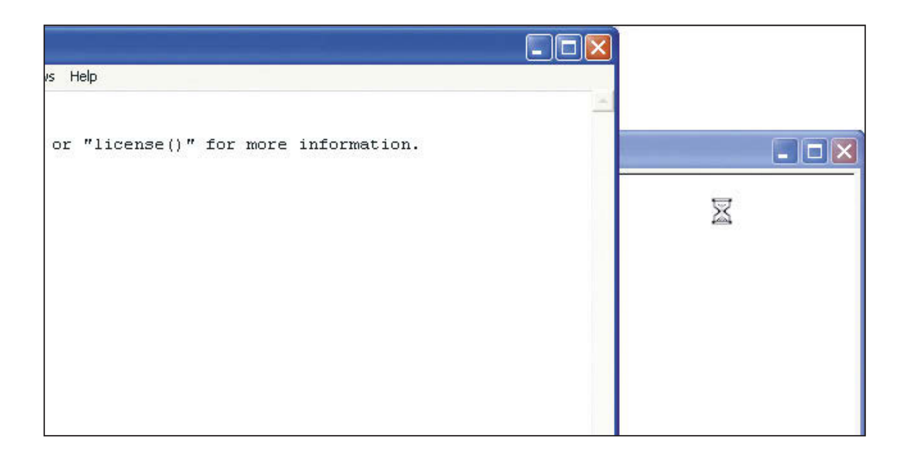
```
>>> t = turtle.Pen()
```

На экране должно появиться белое окно (это холст) со стрелочкой посередине:



Эта стрелочка в центре окна и есть черепашка. Да уж, на черепаху она не слишком-то похожа.

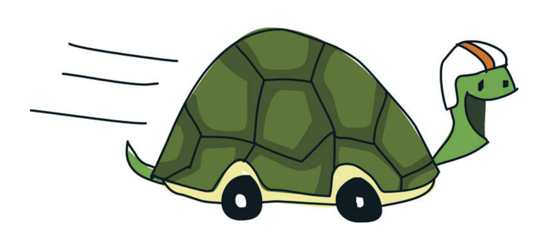
Если окно с черепашкой появится позади окна оболочки Python, могут возникнуть проблемы, и тогда указатель мыши при наведении на окно с черепашкой примет вид песочных часов, как на рисунке:



Такое может произойти по разным причинам: возможно, вы запустили оболочку не кликом по значку на рабочем столе (если вы используете Windows или Mac), а как-то иначе, или, быть может, при установке IDLE произошла ошибка. Закройте оболочку и запустите ее снова, кликнув по значку на рабочем столе. Если это не поможет, попробуйте вместо оболочки воспользоваться консолью Python:

- В Windows выберите **Start ▸ All Programs** (**Пуск ▸ Все программы**), затем в группе **Python 3.2** кликните пункт **Python (command line)**.
- В Mac OS X кликните по значку Spotlight в правом верхнем углу экрана и введите *Terminal*. Когда откроется окно терминала, введите в нем *python*.
- В Ubuntu запустите терминал из меню **Applications** и введите *python*.

Перемещение черепашки

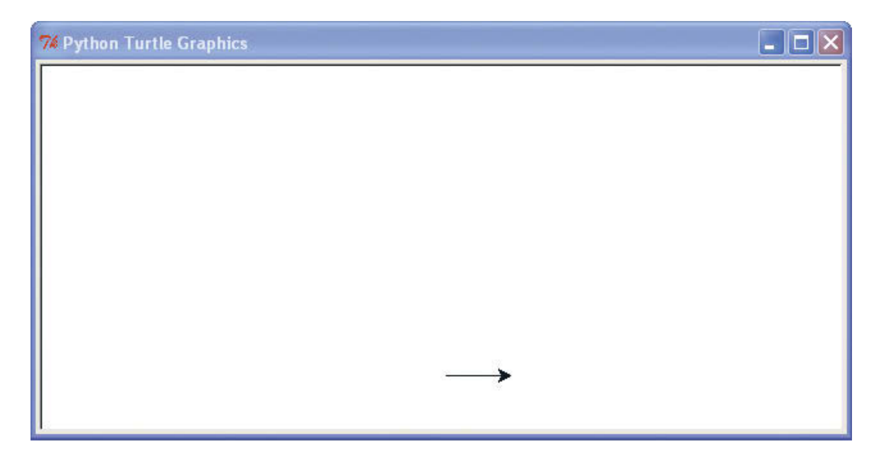


Теперь мы можем давать черепашке команды, вызывая с помощью только что созданной переменной `t` специальные функции, подобно тому как до этого вызывали функцию `Pen` из модуля `turtle`. Например, команда `forward` служит для перемещения черепашки вперед. Чтобы черепашка сдвинулась на 50 пикселей, введите:

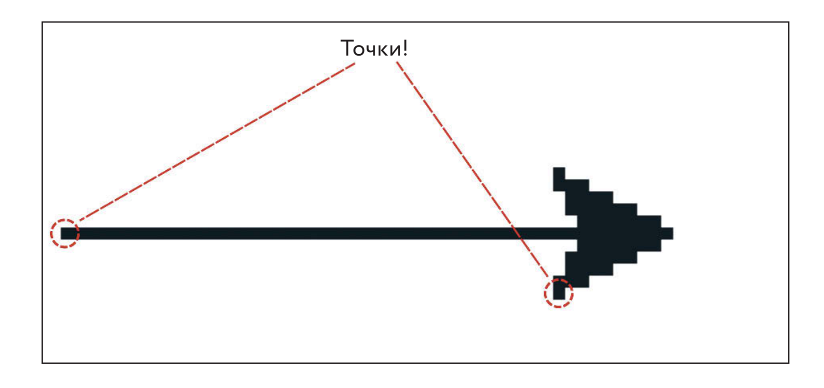
```
>>> t.forward(50)
```

Forward — вперед

Вы должны увидеть нечто подобное:



Черепашка передвинулась на 50 пикселей. *Пиксель* — это одна экранная точка, самый маленький элемент изображения. Все, что вы видите на экране монитора, состоит из пикселей — крошечных квадратных точек. Если посмотреть в увеличении на холст и линию, которую нарисовала черепашка, обнаружится, что и след черепашки, и она сама — просто набор пикселей. Это и есть самая простая компьютерная графика.



Теперь повернем черепашку на 90 градусов влево, введя такую команду:

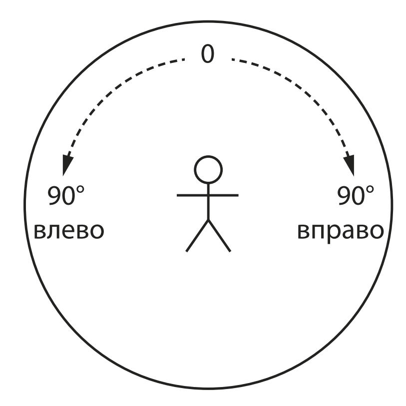
Left — налево

```
>>> t.left(90)
```

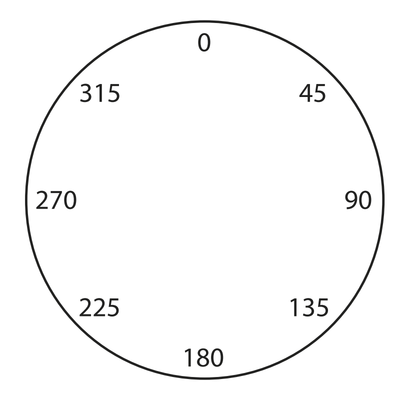
Если вы еще не знаете, что такое градусы, вот как можно это себе представить. Вообразите, что вы стоите в центре круга.

- Направление, в котором вы смотрите, это 0 градусов.
- Если вы вытянете левую руку вбок, это будет 90 градусов влево.
- Если вы вытянете вбок правую руку, это будет 90 градусов вправо.

Вот изображение этих 90-градусных поворотов:

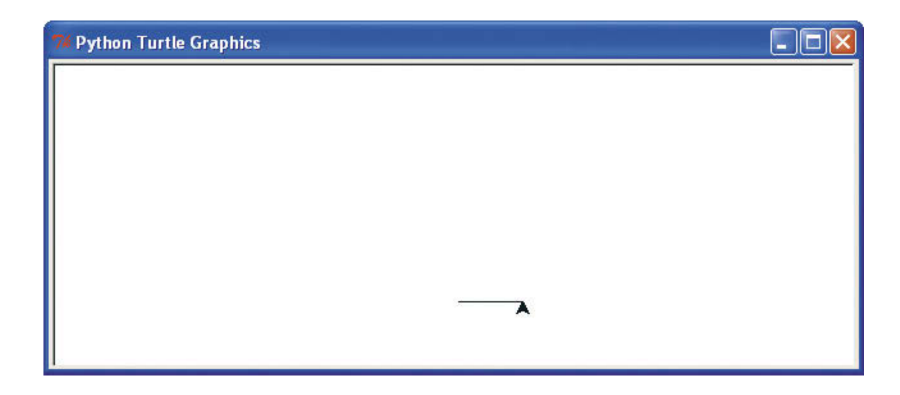


Если продолжать двигаться по часовой стрелке от вашей правой руки и дальше, 180 градусов — это прямо за вашей спиной. 270 градусов — там, куда указывает левая рука, а 360 градусов — направление вашего взгляда, то есть точка, откуда мы начали. Получается, что градусы проходят полный круг от 0 до 360. Вот круг, размеченный на градусы слева направо, с шагом в 45 градусов:



После того как черепашка повернется влево, она будет смотреть в новом направлении — как если бы вы развернулись на 90 градусов туда, куда показывает ваша левая рука.

Поэтому команда `t.left(90)` разворачивает стрелочку острием вверх (так как вначале она указывала вправо):



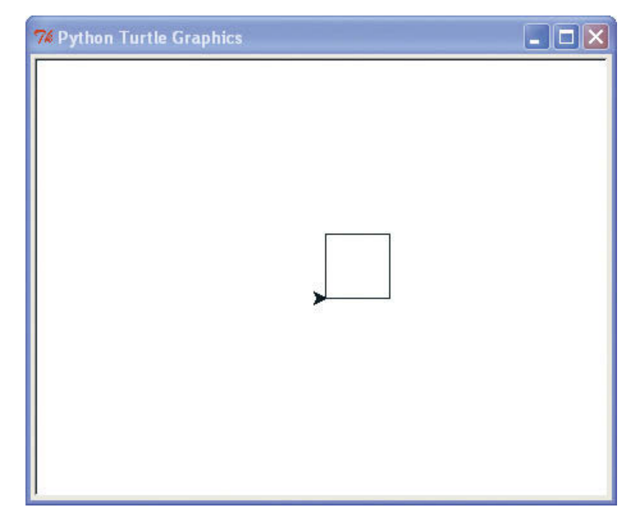
Right — направо

! *Команда* `t.left(90)` *даст тот же результат, что и команда* `t.right(270)`, *а* `t.right(90)` *соответствует* `t.left(270)`. *Просто вообразите себе круг с градусами, и все станет понятно.*

Теперь давайте нарисуем квадрат. Добавьте к уже введенным командам вот такие:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

Черепашка должна изобразить квадрат и развернуться:



Чтобы очистить холст, введите команду reset. При этом с холста исчезнет все нарисованное, а черепашка вернется в начальную позицию.

Reset — сброс

```
>>> t.reset()
```

Либо можно использовать команду clear. Холст также очистится, но черепашка останется на прежнем месте.

```
>>> t.clear()
```

Clear — очистить

Также черепашку можно поворачивать вправо командой right и перемещать назад командой backward. Еще есть команда up, убирающая перо с холста (иными словами, она говорит черепашке, что ей не надо оставлять за собой след, то есть рисовать). С помощью команды down рисование можно снова включить. Вызывать эти функции нужно так же, как и остальные, которыми мы уже пользовались.

Backward — назад
Up — вверх
Down — вниз

Давайте опробуем некоторые из этих команд, создав еще один рисунок. Пусть на этот раз черепашка изобразит две линии. Введите следующий код:

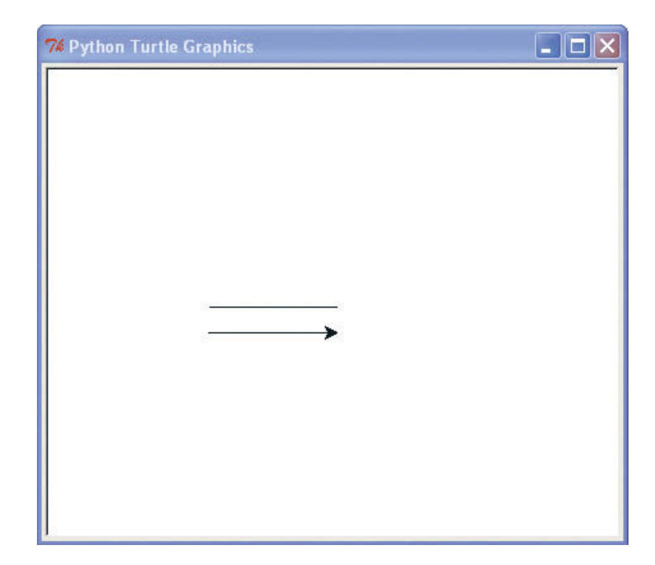
```
>>> t.reset()
>>> t.backward(100)
>>> t.up()
>>> t.right(90)
>>> t.forward(20)
>>> t.left(90)
>>> t.down()
>>> t.forward(100)
```

Сперва командой t.reset() мы очистили холст и переместили черепашку в начальную позицию. Затем с помощью t.backward(100) мы переместили ее на 100 пикселей назад и вызовом t.up() отключили рисование.



Далее командой t.right(90) мы развернули черепашку на 90 градусов вправо так, чтобы она указывала вниз, и командой t.forward(20) передвинули ее на 20 пикселей вперед. При этом на экране ничего не появилось, ведь раньше мы отключили рисование. Затем командой t.left(90) мы повернули черепашку на 90 градусов влево, после чего она стала указывать

вправо, и командой down снова включили рисование. И наконец, командой t.forward(100) мы прочертили линию, параллельную первой. В результате вышел такой рисунок:



Что мы узнали

В этой главе мы узнали, как пользоваться модулем turtle. Мы начертили несколько линий с помощью поворотов влево (left) и вправо (right), а также движений вперед (forward) и назад (backward). Выяснили, что рисование можно отключить командой up и включить командой down. Также мы узнали, что для управления поворотами черепашки используются значения в градусах.

Упражнения

Попробуйте нарисовать с помощью черепашки разные фигуры.

#1. Прямоугольник

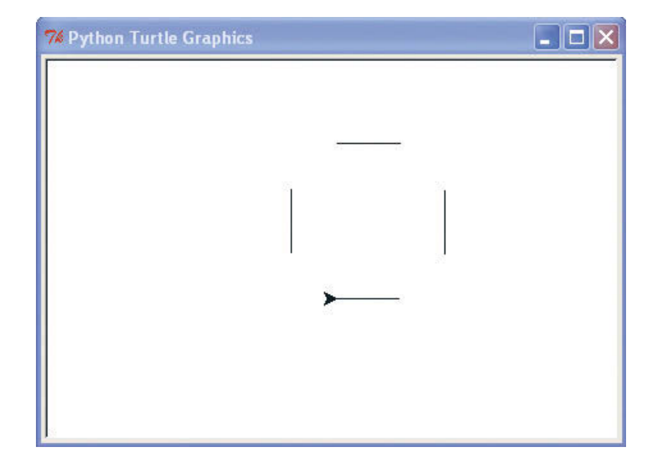
Создайте новый холст с помощью функции `Pen` модуля `turtle` и изобразите на нем прямоугольник.

#2. Треугольник

Создайте новый холст и нарисуйте на нем треугольник. Разворачивая черепашку, сверяйтесь с изображением окружности и градусов поворота (см. «Перемещение черепашки» на стр. 51).

#3. Рамка без углов

Напишите программу, которая рисует четыре линии, как на этом изображении (размер «квадрата» неважен, только форма):



5

ЗАДАЕМ ВОПРОСЫ С ПОМОЩЬЮ IF И ELSE

При написании программ часто приходится задавать вопросы, требующие ответа «да» или «нет», и в зависимости от этого совершать какие-то действия. Например, вы можете спросить: вам больше 20 лет? И если пользователь ответит «да», вывести сообщение: как-то вы староваты!

Подобные вопросы называют *условиями*. В программах на Python условия и ответы обрабатываются с помощью *условной конструкции* `if`. При этом условия могут состоять из более чем одного вопроса и действий тоже может быть много — в зависимости от ответов на каждый заданный вопрос.

В этой главе мы выясним, как использовать конструкцию `if` для написания программ.

Конструкция if

If — если
Age — возраст

Условную конструкцию `if`, которая проверяет возраст, хранящийся в переменной `age`, можно записать, например, так:

```
>>> age = 13
>>> if age > 20:
        print('Как-то вы староваты!')
```

Конструкция `if` состоит из ключевого слова `if`, после которого записано условие, а затем двоеточие, как в строке `if age > 20:`. Следом

за двоеточием должен идти блок команд, и если ответ на вопрос — «да» (в Python это называется Истиной и обозначается словом True), находящиеся в этом блоке команды будут выполнены. А теперь давайте разберемся, как создавать блоки и записывать условия.

True — истина



Блок — это группа команд

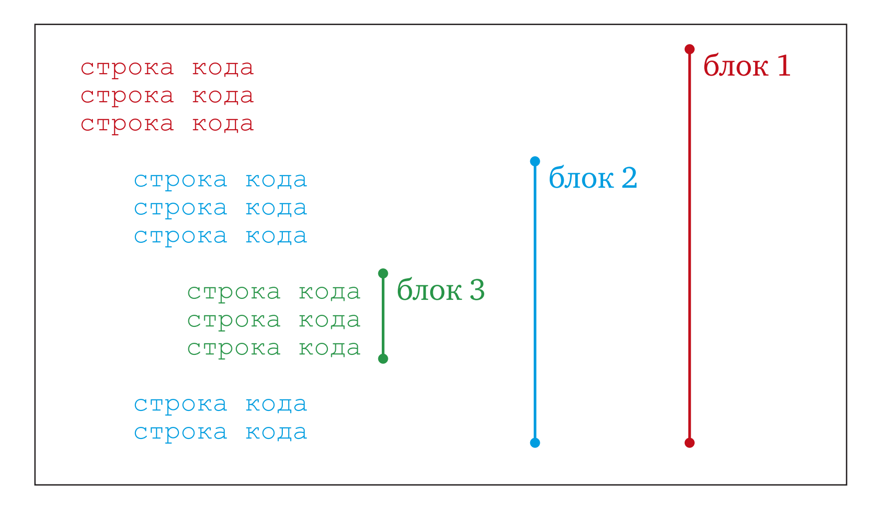
Блок — это набор сгруппированных программных конструкций (команд). Скажем, если условие if age > 20: истинно, может понадобиться выполнить не одно действие (напечатать «Как-то вы староваты!»), а несколько. Например, вывести на экран еще несколько вопросов:

```
>>> age = 25
>>> if age > 20:
        print('Как-то вы староваты!')
        print('Что вы здесь делаете?')
        print('Почему не стрижете газон или не перекладываете ↵
        бумажки?')
```

Здесь блок состоит из трех команд print, которые должны выполняться лишь в том случае, если условие age > 20 окажется истинным. Каждая строка кода в блоке начинается с отступа в четыре пробела (относительно конструкции if, стоящей перед блоком). Давайте посмотрим на этот код еще раз, обозначив пробелы квадратиками:

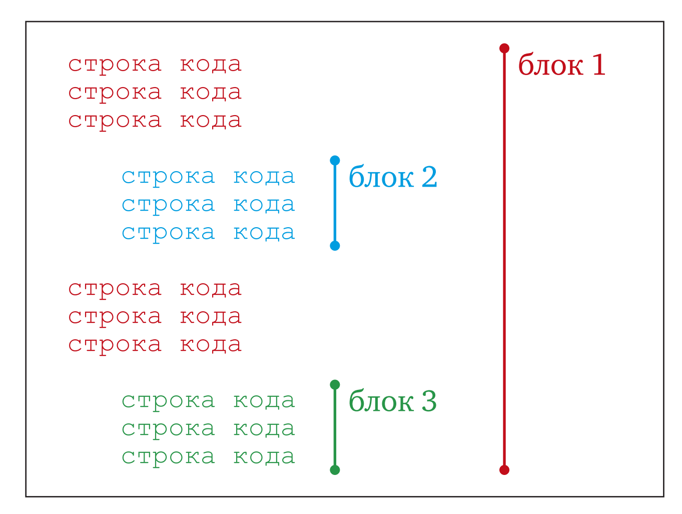
```
>>> age = 25
>>> if age > 20:
    □□□□print('Как-то вы староваты!')
    □□□□print('Что вы здесь делаете?')
    □□□□print('Почему не стрижете газон или не перекладываете ↵
        бумажки?')
```

Пробельные символы, такие как табуляция (которую можно ввести, нажав клавишу **Tab**) или обычные пробелы (вводимые нажатием клавиши **Пробел**), имеют особое значение в языке Python. Строки кода, стоящие в одной позиции (то есть с одинаковым отступом слева), группируются в блок, и каждый раз, когда вы начинаете строку с большего количества пробелов, чем у предыдущей, вы создаете новый блок, являющийся частью предыдущего, как на этом рисунке:



Мы помещаем в блоки команды, которые логически связаны, то есть команды, которые нужно выполнять вместе.

Смена отступа — способ, которым создаются новые блоки. Вот пример трех блоков, которые существуют только благодаря изменению величины отступов:



У блоков 2 и 3 отступы одинаковы, но они считаются разными блоками, поскольку между ними стоит блок с меньшим отступом (меньшим количеством пробелов).

Если одна строка в блоке начинается с четырех пробелов, а следующая с шести, Python при попытке запустить такой код выдаст *ошибку выравнивания*, ведь он ожидает, что для всех строк внутри блока будет использовано одинаковое количество пробелов. Поэтому, если вы начали блок с отступа в четыре пробела, нужно использовать такой же отступ и для остальных строк этого блока. Вот пример:

```
>>> if age > 20:
    □□□□print('Как-то вы староваты!')
    □□□□□□print('Что вы здесь делаете?')
```

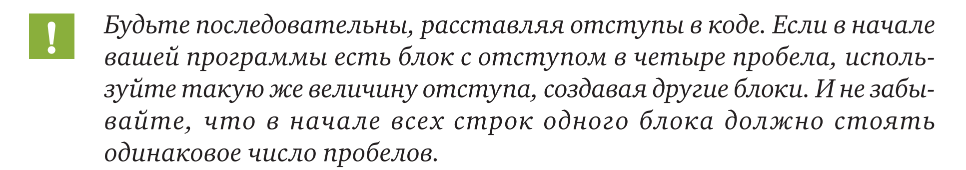
Я изобразил пробелы квадратиками, чтобы показать разницу. Заметьте, третья строка начинается с шести пробелов, а не с четырех.

Если попробовать запустить этот код, IDLE пометит красным проблемное место в строке и выдаст ошибку синтаксиса с пояснением:

```
>>> age = 25
>>> if age > 20:
        print('Как-то вы староваты!')
        ■print('Что вы здесь делаете?')
SyntaxError: unexpected indent
```

Unexpected indent — неожиданное выравнивание

Как видим, Python не ожидал встретить два лишних пробела в начале следующей строки.



Будьте последовательны, расставляя отступы в коде. Если в начале вашей программы есть блок с отступом в четыре пробела, используйте такую же величину отступа, создавая другие блоки. И не забывайте, что в начале всех строк одного блока должно стоять одинаковое число пробелов.

Условия и сравнение значений

Условие — это программная конструкция, которая что-то с чем-то сравнивает, сообщая, является ли заданное соотношение `Истиной` (True) или `Ложью` (False). Например, выражение `age > 10` — это условие, которое как бы задает вопрос: значение переменной `age` больше, чем 10? Вот другое условие: `hair_color == 'лиловый'`, и ему соответствует вопрос: равно ли значение переменной `hair_color` строке `'лиловый'`? Для создания условий в языке Python используются специальные символы-операторы, такие как «равно», «больше» или «меньше». Некоторые из этих операторов показаны в таблице 5.1.

Hair color — цвет волос

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Таблица 5.1. Операторы сравнения

Примеры сравнения: если вам 10 лет, условие `your_age == 10` в результате даст `Истину`, в противном же случае оно даст `Ложь`. Если вам 12 лет, условие `your_age > 10` даст `Истину`.

Your age — ваш возраст



Проверяя два значения на равенство, используйте двойной знак «равно» (==).

Давайте рассмотрим еще несколько примеров. Предположим, вы решили создать условную конструкцию, которая печатает «Вы слишком стары для моих шуток!», если возраст пользователя (`age`) больше 10 лет:



```
>>> age = 10
>>> if age > 10:
        print('Вы слишком стары для моих шуток!')
```

Что случится, если вы введете этот код в IDLE и нажмете на следующей строке **Enter**?

А ничего не случится.

Поскольку значение `age` *не больше* 10, Python не будет выполнять блок с командой `print`. Однако, если вы дадите переменной `age` значение 20, сообщение будет напечатано.

Теперь изменим предыдущий пример, поставив в условии вместо знака «больше» знак «больше или равно» (>=):

```
>>> age = 10
>>> if age >= 10:
        print('Вы слишком стары для моих шуток!')
```

На экране появится сообщение «Вы слишком стары для моих шуток!», ведь `age` равно 10, и сравнение «`age` больше или равно 10» дает `Истину`.

Теперь попробуем сравнение на равенство (==):

```
>>> age = 10
>>> if age == 10:
        print('Что нельзя съесть на завтрак? Обед и ужин!')
```

На экране должно возникнуть сообщение «Что нельзя съесть на завтрак? Обед и ужин!».

Конструкция if-then-else

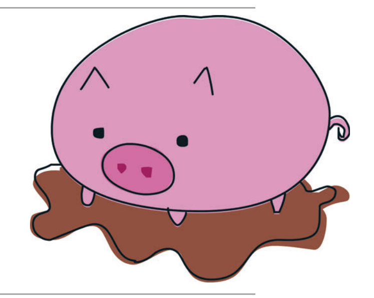
Также с помощью команды `if` можно что-то сделать, если условие не дает `Истину` (то есть дает `Ложь`). Например, если ваш возраст (`age`) равен 12, мы можем вывести на экран одно сообщение, а если не равен 12 — другое. Для этого служит конструкция `if-then-else`, которая работает по принципу: «*если* условие дает `Истину`, сделай это, *иначе* сделай то».

Else — иначе

Давайте опробуем эту конструкцию. Введите в оболочке Python следующее:

```
>>> print("Хотите услышать грязную шутку?")
Хотите услышать грязную шутку?
>>> age = 12
>>> if age == 12:
        print("Свинья шлепнулась в грязь!")
else:
        print("Тсс! Это секрет.")

Свинья шлепнулась в грязь!
```



Поскольку мы задали переменной `age` значение 12, а условие как раз проверяет, равняется ли `age` числу 12, на экране появилось первое сообщение. Теперь поменяйте значение `age` на какое-нибудь другое число, например так:

```
>>> print("Хотите услышать грязную шутку?")
Хотите услышать грязную шутку?
>>> age = 8
>>> if age == 12:
        print("Свинья шлепнулась в грязь!")
else:
        print("Тсс! Это секрет.")

Тсс! Это секрет.
```

На этот раз на экране возникло второе сообщение.

Команды if и elif

Конструкцию `if` можно сделать еще мощнее с помощью ключевого слова `elif` (это сокращение от `else-if`). К примеру, мы можем проверять, чему равен возраст пользователя — 10, 11 или 12 (и так далее), в каждом

Else-if — иначе-если

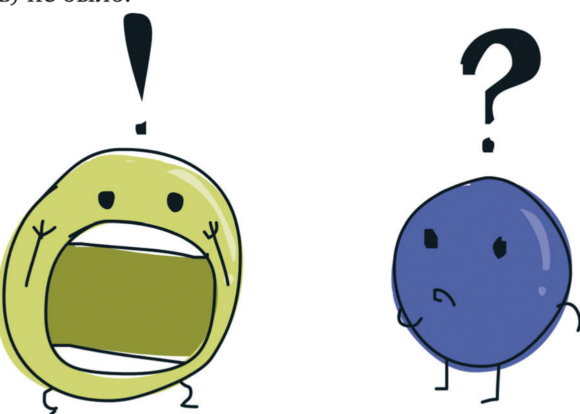
из этих случаев выполняя разные действия. В отличие от else, в одной конструкции if может быть несколько вариантов elif:

```
  >>> age = 12
❶ >>> if age == 10:
❷           print("Что выйдет, если клюква наденет штаны?")
            print("Брюква!")
❸ elif age == 11:
            print("Что сказала зеленая виноградина синей ↵
            виноградине?")
            print("Дыши! Дыши!")
❹ elif age == 12:
❺           print("Что сказал 0 числу 8?")
            print("Привет, ребята!")
  elif age == 13:
            print("Что такое: на потолке сидит и хохочет?")
            print("Муха-хохотуха!")
  else:
            print("Что-что?")

  Что сказал 0 числу 8? Привет, ребята!
```

В строке с отметкой ❶ делается проверка, равно ли значение переменной age числу 10. Соответственно, команды print в блоке ❷ будут выполнены, если age равняется 10. Но поскольку мы задали age значение 12, компьютер переходит на строку ❸, к следующей проверке, где age сравнивается с числом 11. Однако age не равно 11, и компьютер идет дальше, на строку ❹, чтобы проверить, равняется ли age числу 12. На этот раз условие дает Истину, и компьютер выполняет команды print в блоке ❺.

При вводе этого кода IDLE будет ставить отступы автоматически, поэтому не забывайте нажимать **Delete** или **Backspace** после блоков с командами print, так чтобы перед командами if, elif и else отступов (пробелов) не было.



Объединение условий

Несколько условий можно объединить в одно с помощью ключевых слов and (что означает «и») и or (что означает «или»). Вот пример использования or:

```
>>> if age == 10 or age == 11 or age == 12 or age == 13:
        print('13 + 49 + 84 + 155 + 97: что получится? Головная ↵
        боль!')
else:
        print('Что-что?')
```

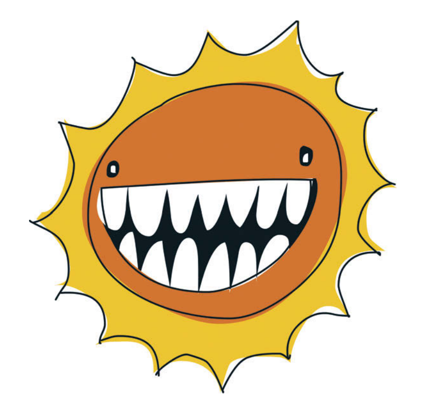
Если любое из условий в первой строке даст Истину (если age равняется 10, 11, 12 *или* 13), будет выполнен блок с командой print на следующей строке.

Если же ни одно из условий не даст Истину, Python перейдет к блоку после **else**, напечатав на экране «Что-что?».

Можно сделать этот пример еще компактнее, воспользовавшись ключевым словом and, а также операторами «больше или равно» (>=) и «меньше или равно» (<=). Вот так:

```
>>> if age >= 10 and age <= 13:
        print('13 + 49 + 84 + 155 + 97: что получится? Головная ↵
        боль!')
else:
        print('Что-что?')
```

Если, согласно условию в первой строке, значение age больше или равно 10 *и* меньше или равно 13, будет выполнен блок с командой print на следующей строке. Например, если значение age равно 12, будет напечатано сообщение «13 + 49 + 84 + 155 + 97: что получится? Головная боль!», ведь 12 больше, чем 10, и меньше, чем 13.



Переменные без значения — None

Можно не только сохранить в переменной число, строку или список, но и назначить переменной пустое значение. В языке Python пустое значение называется None, и оно говорит о том, что переменная ничего не содержит. Обратите внимание, что None — не то же самое, что ноль, поскольку ноль является числом, тогда как None — это отсутствие какого-либо значения. Вот пример:

```
>>> myval = None
>>> print(myval)
None
```

Присвоить переменной значение None значит сказать, что в ней больше ничего не содержится (что эта переменная не связана с каким-либо значением). Еще это способ определить переменную, не указывая ее значение. Делать так стоит, если вы знаете, что эта переменная понадобится позже, и хотите определить все переменные в начале программы. Программисты часто так делают, чтобы имена переменных были на виду.

Проверить переменную на значение None можно с помощью конструкции if. Вот так:

```
>>> myval = None
>>> if myval == None:
        print("В переменной myval ничего нет")

В переменной myval ничего нет
```

Такая проверка пригодится, если вы хотите вычислить некое значение и присвоить его переменной лишь в случае, если в ней еще ничего нет.

Разница между строками и числами

Пользовательским вводом называют данные, которые пользователь вводит с клавиатуры, будь то буква, цифра, нажатие клавиши **Enter** или что-то еще. Пользовательский ввод попадает в Python в виде строки, а значит, если вы напечатаете на клавиатуре 10, Python сохранит эту информацию в переменной как строку, а не как число.

Чем строка '10' отличается от числа 10? Оба значения выглядят одинаково, разница лишь в том, что одно из них в кавычках. Однако для компьютера это разные вещи.

Допустим, мы сравниваем значение переменной age с числом с помощью конструкции if:

```
>>> if age == 10:
        print("Как лучше общаться с монстром?")
        print("Издалека!")
```

Теперь дадим переменной age значение 10:

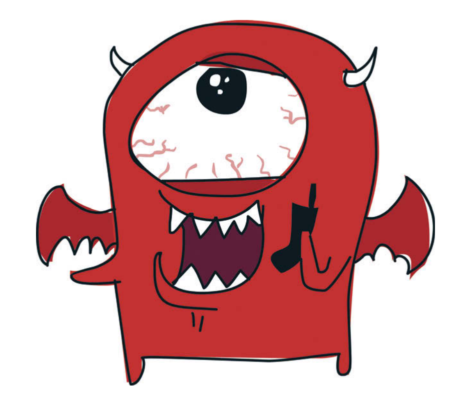
```
>>> age = 10
>>> if age == 10:
        print("Как лучше всего общаться с монстром?")
        print("Издалека!")
Как лучше всего общаться с монстром?
Издалека!
```

Как видите, запустился блок команд print.
Теперь присвоим переменной age строку '10' (в кавычках):

```
>>> age = '10'
>>> if age == 10:
        print("Как лучше всего общаться с монстром?")
        print("Издалека!")
```

В этом случае команды print не сработали, поскольку Python не считает число в кавычках (то есть строку с цифрами) числом.

К счастью, в Python есть волшебные функции, которые превращают строки в числа и числа в строки. Например, сделать из строки '10' число может функция int:



```
>>> age = '10'
>>> converted_age = int(age)
```

После этого в переменной converted_age будет число 10.
А для преобразования числа в строку служит функция str:

Converted — преобразованный

```
>>> age = 10
>>> converted_age = str(age)
```

Теперь значением converted_age будет не число 10, а строка '10'.

Помните команду if age == 10, которая ничего не печатала, когда значением переменной была строка (age='10')? Если сперва мы преобразуем эту строку в число, результат будет другим:

```
>>> age = '10'
>>> converted_age = int(age)
```

```
>>> if converted_age == 10:
        print("Как лучше всего общаться с монстром?")
        print("Издалека!")
Как лучше всего общаться с монстром?
Издалека!
```

Однако будьте осторожны: если вы попытаетесь преобразовать так число с десятичной точкой, произойдет ошибка, поскольку функция `int` ожидает, что в строке будет целое число.

```
>>> age = '10.5'
>>> converted_age = int(age)
Traceback (most recent call last):
File "<pyshell#35>", line 1, in <module>
converted_age = int(age)
ValueError: invalid literal for int() with base 10: '10.5'
```

Value error — ошибка значения

Float — *здесь* «десятичная дробь»

`ValueError` — это тип ошибки, которым Python сообщает, что значение, которое вы попытались использовать, здесь не подходит. Чтобы это исправить, нужно заменить функцию `int` функцией `float`, которая может обрабатывать дробные числа.

```
>>> age = '10.5'
>>> converted_age = float(age)
>>> print(converted_age)
10.5
```

Python выдаст ошибку `ValueError` и в том случае, если в строке, которую вы пытаетесь преобразовать, нет цифровых символов:

```
>>> age = 'десять'
>>> converted_age = int(age)
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
converted_age = int(age)
ValueError: invalid literal for int() with base 10: 'десять'
```

Что мы узнали

В этой главе мы научились работать с конструкциями `if` и создавать блоки кода, которые выполняются, только если условие дает `Истину`. Мы выяснили, что, расширив конструкцию `if` с помощью `elif`, можно выполнять разные блоки кода в зависимости от поставленных условий. Узнали, как использовать ключевое слово `else`, выполняя команды, если ни одно из предыдущих условий не дало `Истину`. Еще мы научились объединять условия с помощью ключевых слов `and` и `or` и так проверять число на вхождение в диапазон, а также преобразовывать строки в числа посредством функций `int`, `str` и `float`. И наконец, мы разобрались, что такое пустое значение `None` и как его использовать для сброса значений переменных.

Упражнения

Решите эти задачи, используя конструкции `if` и условия.

#1. Вы богаты?

Как думаете, что делает этот код? Сначала попробуйте в этом разобраться, не вводя код в оболочку Python, а затем проверьте свой ответ.

Money — деньги

```
>>> money = 2000
>>> if money > 1000:
        print("Я богат!")
else:
        print("Я не богат!")
        print("Может, когда-нибудь потом…")
```

#2. Бисквитики!

Twinkies — печенье с кремовой начинкой

Создайте конструкцию `if`, которая проверяет, действительно ли количество бисквитов (которое задано в переменной `twinkies`) меньше 100 или больше 500. Если это условие выполняется, пусть ваша программа напечатает сообщение «Слишком мало или слишком много».

#3. Подходящая сумма

Создайте конструкцию `if`, которая проверяет, соответствует ли заданная в переменной `money` сумма денег диапазону значений от 100 до 500 *или* диапазону значений от 1000 до 5000.

#4. Я одолею этих ниндзя!

Создайте конструкцию `if`, которая печатает строку «Их слишком много», если количество ниндзя (заданное в переменной `ninjas`) меньше 50, печатает «Будет непросто, но я с ними разделаюсь», если это количество меньше 30, и печатает «Я одолею этих ниндзя!», если количество меньше 10. Проверьте, как ваш код работает с таким значением:

Ninjas — ниндзя

```
>>> ninjas = 5
```

6

ПРИШЛО ВРЕМЯ ЗАЦИКЛИТЬСЯ

Что может быть хуже, чем повторять одно и то же действие снова и снова? Люди не просто так считают овец, когда им не спится, и дело тут не в особых снотворных качествах кудрявых парнокопытных. Суть в том, что без конца заниматься одним и тем же невыносимо скучно, а человеческому мозгу, когда он не сфокусирован на чем-то интересном, проще отключиться.



Программисты тоже не слишком любят однообразные действия (если не страдают бессонницей). К счастью, в большинстве языков программирования есть конструкция под названием «цикл `for`», которая автоматически повторяет другие команды и блоки кода.

В этой главе мы изучим цикл `for`, а также другую разновидность цикла в языке Python — цикл `while`.

Использование цикла for

Если нужно пять раз напечатать слово «привет», вы можете сделать следующее:

```
>>> print("привет")
привет
>>> print("привет")
привет
>>> print("привет")
привет
>>> print("привет")
привет
>>> print("привет")
привет
```

Только уж очень это утомительно. Однако вы можете сократить размер кода (а также количество повторов), воспользовавшись циклом for:

```
❶ >>> for x in range(0, 5):
❷         print('привет')
   привет
   привет
   привет
   привет
   привет
```

Range — диапазон

Функция range в строке ❶ служит для создания списка чисел, который начинается с первого числа в скобках и заканчивается числом на единицу меньше второго числа. Чтобы лучше в этом разобраться, давайте посмотрим, как функция range работает вместе с функцией list. Дело в том, что range не создает готовый список, а возвращает *итератор* — объект языка Python, специально придуманный для работы с циклами. Однако, если передать этот итератор функции list, получится список чисел.

```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Если вернуться к нашему циклу for, код в строке ❶ будет представлять собой такую инструкцию:

- Начать отсчет с 0 и остановиться, не доходя до числа 5.
- Каждое отсчитанное значение сохранять в переменной x.

Далее Python исполняет блок кода ❷. Обратите внимание, что этот блок начинается с отступа в 4 пробела. При вводе программы IDLE ставит этот отступ автоматически.

После ввода последней строки и нажатия **Enter** программа напечатает слово «привет» пять раз.

Переменную x можно использовать в команде print для подсчета «приветов»:

```
>>> for x in range(0, 5):
        print('привет %s' % x)
привет 0
привет 1
привет 2
привет 3
привет 4
```

Если записать этот код без цикла for, он будет выглядеть примерно так:

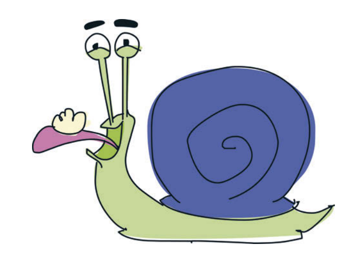
```
>>> x = 0
>>> print('привет %s' % x)
привет 0
>>> x = 1
>>> print('привет %s' % x)
привет 1
>>> x = 2
>>> print('привет %s' % x)
привет 2
>>> x = 3
>>> print('привет %s' % x)
привет 3
>>> x = 4
>>> print('привет %s' % x)
привет 4
```

Выходит, цикл избавил нас от необходимости писать восемь дополнительных строк кода. Хорошие программисты терпеть не могут делать одно и то же несколько раз, поэтому цикл — одна из самых популярных программных конструкций.

При создании цикла for не обязательно использовать функцию range или list. Вместо этого можно воспользоваться созданным ранее списком, таким как список ингредиентов из главы 3:

```
>>> wizard_list = ['паучьи лапки', 'жабий палец', 'язык улитки',
        'крыло летучей мыши', 'жир слизня', 'медвежий коготь']
>>> for i in wizard_list:
        print(i)
паучьи лапки
жабий палец
язык улитки
крыло летучей мыши
жир слизня
медвежий коготь
```

Этот код означает следующее: «Для каждого значения из списка wizard_list: сохранить значение в переменной i и вывести значение этой переменной на экран». Без цикла for нам бы пришлось написать что-то вроде такого:



```
>>> wizard_list = ['паучьи лапки', 'жабий палец', 'язык улитки',
'крыло летучей мыши', 'жир слизня', 'медвежий коготь']
>>> print(wizard_list[0])
паучьи лапки
>>> print(wizard_list[1])
жабий палец
>>> print(wizard_list[2])
язык улитки
>>> print(wizard_list[3])
крыло летучей мыши
>>> print(wizard_list[4])
жир слизня
>>> print(wizard_list[5])
медвежий коготь
```

И снова цикл сэкономил нам уйму времени и сил.

Давайте создадим еще один цикл. Введите в оболочке следующий код (отступы IDLE поставит автоматически).

Huge hairy pants — огромные волосатые штаны

```
❶ >>> hugehairypants = ['огромные', 'волосатые', 'штаны']
❷ >>> for i in hugehairypants:
❸           print(i)
❹           print(i)
❺
❻ огромные
  огромные
  волосатые
  волосатые
  штаны
  штаны
```



В строке программы с меткой ❶ мы создаем список, содержащий строки `'огромные'`, `'волосатые'` и `'штаны'`. В строке ❷ перебираем элементы этого списка, каждый раз присваивая очередное значение переменной `i`. В строках ❸ и ❹ дважды выводим содержимое `i` на экран. Нажатие **Enter** в строке ❺ отмечает конец блока, после чего Python запускает введенный ранее код, дважды печатая каждый элемент списка.

Не забывайте, что, введя некорректное число пробелов, вы получите сообщение об ошибке. Например, если вы поставите в начале строки ❹ лишний пробел, Python выдаст ошибку выравнивания:

```
>>> hugehairypants = ['огромные', 'волосатые', 'штаны']
>>> for i in hugehairypants:
        print(i)
         print(i)

SyntaxError: unexpected indent
```

Как мы уже знаем из главы 5, Python ожидает, что у всех строк в блоке будут одинаковые отступы. И неважно, из скольких пробелов состоит отступ, главное, чтобы каждая новая строка блока начиналась с такого же отступа, как и предыдущая (такой код легче для человеческого восприятия).

Вот более сложный пример цикла с двумя блоками кода:

```
>>> hugehairypants = ['огромные', 'волосатые', 'штаны']
>>> for i in hugehairypants:
        print(i)
        for j in hugehairypants:
            print(j)
```

Где в этом коде блоки? Первый блок — содержимое первого цикла for:

```
>>> hugehairypants = ['огромные', 'волосатые', 'штаны']
>>> for i in hugehairypants:
      print(i)                  #
      for j in hugehairypants: # Эти строки — в ПЕРВОМ блоке
            print(j)            #
```

Второй блок состоит из единственной команды print во втором цикле for:

```
❶ hugehairypants = ['огромные', 'волосатые', 'штаны']
  for i in hugehairypants:
          print(i)
❷         for j in hugehairypants:
❸               print(j) # Эта строка — также и во ВТОРОМ блоке
```

Постарайтесь понять, что напечатает эта небольшая программа.

В строке ❶ создается список hugehairypants, и, глядя на следующие две строки, можно заключить, что цикл переберет все элементы списка, напечатав каждый из них. Однако в строке ❷ начинается еще один цикл по тому же списку, причем на этот раз элементы попадают в переменную j, и далее в строке ❸ каждое значение будет снова напечатано. Строки ❷ и ❸ все еще являются частью первого цикла, а значит, они будут выполнены для каждого элемента списка по мере их перебора в первом цикле for, то есть для каждого значения переменной i.

Запустив этот код, вы увидите слово «огромные», затем слова «огромные», «волосатые», «штаны», потом «волосатые» и следом «огромные», «волосатые», «штаны» и так далее.

Введите этот код в оболочке Python и убедитесь сами:

```
  >>> hugehairypants = ['огромные', 'волосатые', 'штаны']
  >>> for i in hugehairypants:
❶         print(i)
          for j in hugehairypants:
❷             print(j)

◈ огромные
  огромные
  волосатые
  штаны
◈ волосатые
  огромные
  волосатые
  штаны
◈ штаны
  огромные
  волосатые
  штаны
```

В строке с меткой ❶ Python заходит в первый цикл и печатает первый элемент списка. Затем заходит во второй цикл, где, перебирая список, печатает в строке ❷ все его элементы. После этого снова переходит к команде `print(i)`, печатая второй элемент списка, опять печатает весь список с помощью второго цикла и `print(j)`, затем то же самое повторяется для третьего элемента. Значком ◈ я пометил в выводе программы строки, которые печатает команда `print(i)`, строки же без отметок печатает `print(j)`.

А теперь, как насчет чего-нибудь более полезного, чем печать каких-то словечек?

Помните вычисления из главы 2, где мы составили выражение для количества золотых монет, которое вы накопите к концу года, пользуясь копировальным агрегатом дедушки? Выглядело это выражение так:

```
>>> 20 + 10 * 365 – 3 * 52
```



Здесь учитываются 20 найденных монет плюс 10 волшебных монет, умноженных на 365 дней в году, минус 3 монеты в неделю, которые крадет ворона.

Любопытно посмотреть, как будут расти ваши богатства с каждой неделей. Для этого можно использовать цикл for, но сначала стоит изменить значение переменной magic_coins, чтобы в ней хранилось общее количество волшебных монет, полученных за неделю. Нужно 10 волшебных монет в день умножить на 7 дней в неделе, то есть magic_coins будет равняться 70:

Magic coins — волшебные монеты

```
>>> found_coins = 20
>>> magic_coins = 70
>>> stolen_coins = 3
```

Теперь можно полюбоваться, как приумножаются ваши сокровища. Для этого понадобится еще одна переменная — coins, а также цикл:

```
   >>> found_coins = 20
   >>> magic_coins = 70
   >>> stolen_coins = 3
❶ >>> coins = found_coins
❷ >>> for week in range(1, 53):
❸         coins = coins + magic_coins — stolen_coins
❹         print('Неделя %s = %s' % (week, coins))
```

В строке ❶ мы присваиваем переменной coins значение переменной found_coins (найденные монеты, ваш стартовый капитал). В строке ❷ начинается цикл for, который будет выполнять команды из блока, состоящего из строк ❸ и ❹. При каждом повторе цикла в переменную week будет попадать номер очередной недели — с первой по 52-ю.

Week — неделя

Строка ❸ посложнее. Для каждой недели нам нужно добавлять количество монет, созданных волшебным образом, и вычитать монеты, украденные вороной. Представьте, что переменная coins — это нечто вроде сундука с сокровищами, в который каждую неделю попадают новые монеты. Так что эта строка означает: заменить значение переменной coins суммой ее нынешнего значения и количества монет, полученных за неделю. Знак «равно» здесь словно указывает: посчитать то, что справа, и сохранить полученное значение в стоящей слева переменной.

Строка ❹ состоит из команды print, которая с помощью строки с метками %s печатает номер недели и количество монет, накопленных на данный момент (если вам непонятно, о чем я, перечитайте раздел «Переменные внутри строк» на стр. 36). Итак, запустив программу, вы увидите что-то вроде:

```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> found_coins = 20
>>> magic_coins = 70
>>> stolen_coins = 3
>>> coins = found_coins
>>> for week in range(1, 53):
        coins = coins + magic_coins - stolen_coins
        print('Неделя %s = %s' % (week, coins))


Неделя 1 = 87
Неделя 2 = 154
Неделя 3 = 221
Неделя 4 = 288
Неделя 5 = 355
Неделя 6 = 422
Неделя 7 = 489
Неделя 8 = 556
Неделя 9 = 623
Неделя 10 = 690
Неделя 11 = 757
Неделя 12 = 824
Неделя 13 = 891
Неделя 14 = 958
Неделя 15 = 1025
Неделя 16 = 1092
Неделя 17 = 1159
Неделя 18 = 1226
Ln: 64  Col: 4
```

Цикл while

Цикл for — не единственный вид циклов в языке Python. Есть также цикл while, который используется, если нужное количество повторов заранее неизвестно, тогда как for расчитан на определенное число повторов.

Представьте себе лестницу с 20 ступеньками. Это внутренняя лестница в помещении, и вы знаете, что запросто преодолеете ступеньки. Таков и цикл for.

Step — шаг

```
>>> for step in range(0, 20):
        print(step)
```

А теперь вообразите лестницу на горе. Гора высокая, и вы можете лишиться сил прежде, чем доберетесь до вершины. К тому же погода может испортиться, и тогда придется прекратить подъем. На такую лестницу похож цикл while.

```
step = 0
while step < 10000:
        print(step)
        if tired == True:
              break
        elif badweather == True:
              break
        else:
              step = step + 1
```

Tired — уставший

Break — перерыв

Bad weather — плохая погода

Если вы попробуете запустить этот код, возникнет ошибка. Почему? Потому что мы не создали переменные `tired` (признак усталости) и `badweather` (признак плохой погоды). Для работающей программы здесь не хватает кода, зато это наглядный пример использования цикла `while`.



Сперва мы создаем переменную `step` (ступенька), присваивая ей значение 0. Затем — цикл `while` с условием, которое проверяет, меньше ли значение `step` 10000 (`step < 10000`), это полное количество ступеней от подножия до вершины горы. И до тех пор, пока это условие будет давать `Истину`, остальные строки программы будут выполняться циклически.

Команда `print(step)` выводит номер ступеньки, далее мы с помощью `if` проверяем значение `tired` на равенство `True` (`True` в языке Python обозначает `Истину` и является булевым значением; что это такое, мы выясним в главе 8.) Если `tired == True`, мы выходим из цикла с помощью команды `break`. Команда `break` — это способ немедленного выхода из цикла (иными словами, его завершение), который подходит как для `while`, так и для `for` (при этом программа продолжает выполняться со строк, идущих после цикла и его блока, впрочем, в нашем примере таких строк нет).

В строке `elif badweather == True`: переменная `badweather` проверяется на равенство `True`. Если это так, команда `break` завершит цикл. И наконец, если ни `tired`, ни `badweather` не равны `True`, выполнится строчка после `else`, где мы прибавим 1 к значению `step` (`step = step + 1`), после чего цикл перейдет к следующему повтору.

Итак, цикл `while` выполняет следующие действия:

1. Проверяет условие.

2. Выполняет код в блоке.

3. Повторяет все сначала.

Часто в цикле `while` используют не одно условие, а несколько. Например, так:

```
❶ >>> x = 45
❷ >>> y = 80
❸ >>> while x < 50 and y < 100:
          x = x + 1
          y = y + 1
          print(x, y)
```

And — и

В строке ❶ мы создаем переменную `x`, дав ей значение 45, а в строке ❷ — переменную `y` со значением 80. Цикл в строке ❸ проверяет два условия: верно ли, что `x` меньше 50, *и* верно ли, что `y` меньше 100. До тех пор, пока оба условия дают `Истину`, будут выполняться следующие строки, где каждая из переменных увеличивается на 1, а затем значения `x` и `y` выводятся на экран.

```
46 81
47 82
48 83
49 84
50 85
```

Понимаете, что здесь происходит?

Мы начинаем считать со значения 45 для `x` и со значения 80 для `y`, а затем увеличиваем их на 1 при каждом повторе цикла. Цикл будет выполняться, пока `x` меньше 50, а `y` меньше 100. После пяти повторов (при каждом из которых каждая из двух переменных увеличивалась на 1) значение `x` достигло 50. Первое условие (`x < 50`) перестало быть истинным, и Python завершил цикл.

Еще цикл `while` часто применяется в качестве условно бесконечного цикла, который выполняется до тех пор, пока код внутри него не завершит цикл командой `break`. Примерно так:

```
while True:
        много кода
        много кода
        много кода
        if some_value == True:
            break
```

Some value — какое-нибудь значение

В условии цикла `while` стоит просто `True`, то есть `Истина`, а значит, код внутри блока будет выполняться всегда, поэтому цикл

бесконечный. Он будет завершен с помощью `break`, если переменная `some_value` примет значение `True`. Более удачный пример такого подхода есть в разделе «Использование `randint`» на стр. 133, но прежде чем его изучать, стоит прочесть главу 7.

Что мы узнали

В этой главе мы научились использовать для повторяющихся задач циклы, чтобы избежать утомительных вводов вручную. Группировали команды, которые нужно дублировать, в блоки кода и помещали их в циклы. Применяли два вида циклов: `for` и `while`. Они похожи, но используются по-разному. Также мы узнали, что ключевое слово `break` служит для принудительного завершения циклов.

Упражнения

Вот несколько задач на использование циклов для самостоятельного выполнения.

#1. Цикл с приветом

Как вы считаете, что делает эта программа? Сперва придумайте вариант ответа, а потом запустите код и проверьте, угадали ли вы.

```
>>> for x in range(0, 20):
        print('привет %s' % x)
        if x < 9:
            break
```

#2. Четные числа

Создайте цикл, который печатает четные числа до тех пор, пока не выведет ваш возраст. Если ваш возраст — нечетное число, создайте цикл, который печатает нечетные числа до совпадения с возрастом. Программа должна выводить на экран нечто подобное:

```
2
4
6
8
10
12
14
```

#3. Пять любимых ингредиентов

Создайте список с пятью разными ингредиентами для бутерброда, наподобие:

Ingredients — ингредиенты

```
>>> ingredients = ['слизни', 'пиявки', 'катышки из пупка гориллы',
        'брови гусеницы', 'пальцы многоножки']
```

Теперь создайте цикл, который печатает список ингредиентов с нумерацией:

```
1 слизни
2 пиявки
3 катышки из пупка гориллы
```

```
4 брови гусеницы
5 пальцы многоножки
```

#4. Ваш лунный вес

Если бы вы сейчас были на Луне, ваш вес составил бы 16,5 процентов от земного. Чтобы узнать, сколько это, умножьте свой земной вес на 0,165.

Если бы каждый год в течение следующих 15 лет вы прибавляли по одному килограмму веса, каким бы оказался ваш лунный вес в каждый из ежегодных визитов на Луну вплоть до 15-го года? Напишите программу, которая с помощью цикла `for` печатает на экране ваш лунный вес в каждом году.

7

ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ КОДА С ПОМОЩЬЮ ФУНКЦИЙ И МОДУЛЕЙ

Подумайте о мусоре, который вы изо дня в день выкидываете на помойку: бутылках и банках, пакетах из-под чипсов, оберточной бумаге, газетах, журналах и так далее. А теперь представьте, что будет, если свалить весь этот мусор огромной кучей на дороге перед вашим домом.

Наверное, вы постараетесь как-то избавиться от этого хлама, потому что никому не нравится преодолевать мусорные горы по дороге в школу. Если отправить все это на переработку, мы сможем заново использовать вещи, которые просто загрязняли бы планету.



В мире программирования повторное использование также играет очень важную роль. Конечно, ваша программа не будет валяться на свалке. Однако, если не применять в новых программах части старого кода, со временем пальцы сточатся о клавиатуру. Кроме того, повторное использование кода делает программы короче и понятнее.

Python предлагает несколько способов повторного использования кода.

Применение функций

С некоторыми случаями повторного использования кода вы уже знакомы. Например, в прошлой главе с помощью функций `range` и `list` мы получили список с последовательно идущими значениями.

```
>>> list(range(0, 5))
[0,1,2,3,4]
```

Конечно, не проблема создать подобный список вручную, но чем длиннее список, тем больше символов придется вводить. Однако с помощью функций можно за один раз создать список в тысячу чисел.

Вот пример создания такого списка при помощи функций `list` и `range`:

```
>>> list(range(0, 1000))
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16...,997,998,999]
```

Функция — это фрагмент кода, выполняющий ту или иную задачу. Кроме того, это один из способов повторного использования кода, ведь одну и ту же функцию можно вызывать в своих программах снова и снова.

Функции полезны при написании простых программ, а при создании программ более сложных, таких как компьютерные игры, функции просто *необходимы* — конечно, если вы планируете закончить программу быстрее, чем за сто лет.

Строение функции

Функция состоит из трех частей: *имени*, *аргументов* и *тела*. Вот пример простой функции:

```
>>> def testfunc(myname):
        print('Привет, %s' % myname)
```

Имя этой функции — `testfunc`. У нее есть единственный аргумент — `myname`, а ее тело — это блок кода, идущий сразу после строки, которая начинается с `def` (сокращение от define — определить). Аргумент — это специальная переменная, которая существует, только пока функция выполняется.

Функцию `testfunc` можно запустить (или, как говорят программисты, *вызвать*), указав ее имя, а после него, в скобках, значение аргумента:

Test func — от test function — тестовая функция
My name — мое имя
Def — от define — определить

```
>>> testfunc('Мэри')
Привет, Мэри
```

Также можно создать функцию, которая принимает не один, а два, три или иное количество аргументов (либо, напротив, не принимает аргументы). Например, вот функция `testfunc` с двумя аргументами:

```
>>> def testfunc(fname, lname):
        print('Привет, %s %s' % (fname, lname))
```

При вызове функции значения аргументов следует писать через запятую:

```
>>> testfunc('Мэри', 'Смит')
Привет, Мэри Смит
```

В качестве аргументов можно указывать переменные (которые перед этим нужно создать):

First name — имя
Last name — фамилия

```
>>> firstname = 'Джо'
>>> lastname = 'Робертсон'
>>> testfunc(firstname, lastname)
Привет, Джо Робертсон
```

Return — вернуть

Зачастую из функции полезно вернуть какое-то значение. Это делается с помощью команды `return`. Например, представьте такую функцию для расчета сбережений:

Savings — сбережения

```
>>> def savings(pocket_money, paper_route, spending):
        return pocket_money + paper_route - spending
```

Pocket money — карманные деньги
Paper route — выручка за доставку газет
Spending — расходы

Эта функция принимает три аргумента. Вычисляя результат, она складывает первые два аргумента — `pocket_money` и `paper_route`, — а затем вычитает из полученного значения третий аргумент — `spending`. Команда `return` возвращает результат вычислений в ту часть кода, откуда функция была вызвана, и его можно сохранить в переменной (обычным способом, с помощью знака "=") либо вывести на экран:

```
>>> print(savings(10, 10, 5))
15
```

Переменные и область видимости

Переменные, созданные в теле функции, нельзя использовать после того, как эта функция завершит работу, поскольку они существуют только во время ее выполнения. В таких случаях программисты говорят, что *область видимости* переменных ограничена функцией.

Рассмотрим простую функцию, которая использует внутренние переменные, но не принимает аргументы:

```
❶ >>> def variable_test():
          first_variable = 10
          second_variable = 20
❷         return first_variable * second_variable
```

Variable test — тестирование переменной
First variable — первая переменная
Second variable — вторая переменная

Здесь в строке ❶ мы создали функцию с именем variable_test, которая в строке ❷ перемножает переменные first_variable и second_variable и возвращает результат.

```
>>> print(variable_test())
200
```

Вызвав эту функцию с помощью print, мы получили результат: 200. Однако, если попытаться за пределами функции вывести на экран значение first_variable (или second_variable), Python выдаст сообщение об ошибке:

```
>>> print(first_variable)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    print(first_variable)
NameError: name 'first_variable' is not defined
```

У переменных, созданных вне тела функции, область видимости другая. Например, давайте создадим перед определением нашей функции переменную, а затем используем ее в теле функции:

```
❶ >>> another_variable = 100
  >>> def variable_test2():
          first_variable = 10
          second_variable = 20
❷         return first_variable * second_variable * another_variable
```

Another variable — другая переменная

Хотя переменные first_variable и second_variable вне тела функции недоступны, переменную another_variable (созданную за пределами функции — в строке ❶) можно использовать и в теле функции — в строке ❷.

Вот что получится при вызове этой функции:

```
>>> print(variable_test2())
20000
```



Теперь представьте, что вы строите звездолет из экономичных материалов, например банок из-под газировки. Вы планируете сооружать его постепенно, расплющивая по 2 банки в неделю, а всего потребуется около 500 банок. Мы можем написать функцию для расчета времени, которое понадобится, чтобы расплющить 500 банок, если считать, что в неделю вы справляетесь с двумя.

Пусть наша функция выводит общее количество расплющенных банок для каждой недели от начала строительства звездолета в течение года. Количество банок в неделю мы будем передавать в аргументе:

Spaceship building — строительство космического корабля
Cans — банки
Total cans — общее количество банок
Week — неделя

```
>>> def spaceship_building(cans):
        total_cans = 0
        for week in range(1, 53):
            total_cans = total_cans + cans
            print('Неделя %s, банок: %s' % (week, total_cans))
```

В первой строке этой функции мы создаем переменную total_cans, присваивая ей значение 0. Затем мы делаем цикл по неделям в году, на каждом шаге прибавляя к total_cans количество банок, расплющенных за неделю. Весь этот блок кода и есть тело нашей функции. При этом в функции есть еще один блок, находящийся внутри первого: это две последние строки кода, составляющие тело цикла for.

Введем код этой функции в окне оболочки и попробуем вызывать ее с разными значениями аргумента (для разного количества расплющенных банок в неделю):

```
>>> spaceship_building(2)

Неделя 1, банок: 2
Неделя 2, банок: 4
Неделя 3, банок: 6
Неделя 4, банок: 8
```

```
Неделя 5, банок: 10
Неделя 6, банок: 12
Неделя 7, банок: 14
Неделя 8, банок: 16
Неделя 9, банок: 18
Неделя 10, банок: 20
(и так далее...)

>>> spaceship_building(13)
Неделя 1, банок: 13
Неделя 2, банок: 26
Неделя 3, банок: 39
Неделя 4, банок: 52
Неделя 5, банок: 65
(и так далее...)
```

Эту функцию можно вызывать снова и снова для разного количества банок в неделю, что гораздо удобнее, чем вводить цикл for каждый раз, когда вам понадобится произвести расчеты с другими параметрами.

Кроме того, функции можно сгруппировать в модуль, и вот тут-то Python по-настоящему показывает свою мощь.

Применение модулей

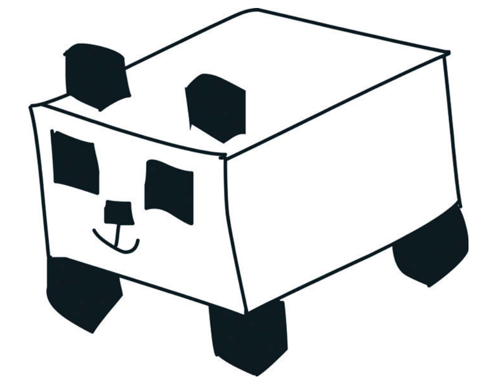
Модули нужны для группировки функций, переменных и других фрагментов Python-кода. Одни модули идут в комплекте с Python, а другие нужно скачивать из интернета. Существуют модули, которые помогают создавать игры (например, идущий с Python tkinter или PyGame, который нужно отдельно скачивать), модули для работы с изображениями (например, PIL) и модули для трехмерной графики (к примеру, Panda3D).

tkinter — от Toolkit interface — интерфейс инструментов разработки

С помощью модулей можно решать самые разные задачи. Например, если вы разрабатываете игру-симулятор и хотите, чтобы игровой мир реалистично изменялся, можете использовать для расчета текущей даты и времени модуль time:

Time — время

```
>>> import time
```



Здесь команда import указывает Python, что мы собираемся работать с модулем time.

Теперь можно через символ «точка» обращаться к функциям, которые содержатся в этом модуле. (Вспомните вызовы функций для работы с черепашкой из главы 4, такие как t.forward(50)). Например, вызвать функцию asctime из модуля time можно следующим образом:

```
>>> print(time.asctime())
'Mon Nov 5 12:40:27 2016'
```



Функция asctime — часть модуля time, которая возвращает текущие дату и время в виде строки.

А теперь представьте, что у пользователя вашей программы нужно запросить некое значение, скажем, возраст или дату рождения. Для этого можно воспользоваться командой print для вывода сообщения с вопросом, а также модулем sys, где находятся инструменты для взаимодействия с интерпретатором Python. Первым делом нужно импортировать модуль sys:

Sys — от system — система

```
>>> import sys
```

Stdin — от standard input — стандартный ввод

Read line — прочесть строку

В модуле sys имеется особый объект под названием stdin, для которого задана очень полезная функция — readline. Эта функция считывает строку набранного на клавиатуре текста вплоть до нажатия **Enter**. (О том, как устроены объекты, мы поговорим в главе 8.) Давайте проверим работу readline, введя в оболочке следующий код:

```
>>> import sys
>>> print(sys.stdin.readline())
```

Если теперь ввести какой-нибудь текст и нажать **Enter**, этот текст отобразится в оболочке.

Вернемся к примеру использования конструкции if из главы 5:

Age — возраст

```
>>> if age >= 10 and age <= 13:
        print('13 + 49 + 84 + 155 + 97: что получится? Головная ↵
        боль!')
else:
        print('Что-что?')
```

Теперь, вместо того чтобы прописывать значение age перед конструкцией if, можно запросить это значение у пользователя. Однако сперва давайте поместим наш код в функцию:

Silly age joke — глупая шутка про возраст

```
>>> def silly_age_joke(age):
        if age >= 10 and age <= 13:
```

```
        print('13 + 49 + 84 + 155 + 97: что получится?
        Головная боль!')
    else:
        print('Что-что?')
```

Теперь нашу функцию можно вызывать, указывая ее имя, а после него, в скобках, значение аргумента `age`:

```
>>> silly_age_joke(9)
Что-что?
>>> silly_age_joke(10)
13 + 49 + 84 + 155 + 97: что получится? Головная боль!
```

Ура, работает! Теперь изменим функцию так, чтобы она запрашивала возраст у пользователя. Менять код функций или добавлять в них новый код можно сколько угодно раз.

```
>>> def silly_age_joke():
        print('Сколько вам лет?')
❶       age = int(sys.stdin.readline())
❷       if age >= 10 and age <= 13:
            print('13 + 49 + 84 + 155 + 97: что получится? ↵
            Головная боль!')
        else:
            print('Что-что?')
```

Посмотрите на строку с меткой ❶. Узнаете функцию, которая преобразует строку текста в число? Мы используем ее здесь, поскольку `readline()` возвращает введенные данные в виде строки, а нам требуется число, чтобы в строке ❷ сравнить его со значениями 10 и 13. Теперь проверим нашу функцию: вызовем ее без параметров, и после того, как на экране появится "Сколько вам лет? ", введем какое-нибудь число:

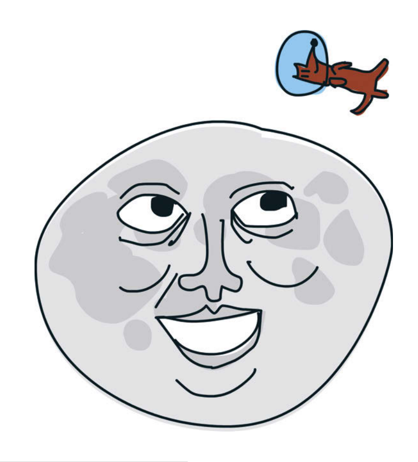
```
>>> silly_age_joke()
Сколько вам лет?
10
13 + 49 + 84 + 155 + 97: что получится? Головная боль!
>>> silly_age_joke()
Сколько вам лет?
15
Что-что?
```

Что мы узнали

В этой главе мы выяснили, что фрагменты кода можно использовать повторно с помощью функций, и узнали, как вызывать функции, которые содержатся в модулях. Рассмотрели, как область видимости переменных влияет на возможность использовать их внутри функции и вне ее, и научились создавать функции с помощью ключевого слова `def`. Также мы научились импортировать модули, чтобы использовать код, который в них содержится.

Упражнения

Вот несколько упражнений по созданию ваших собственных функций.



#1. Функция лунного веса

Одним из заданий к главе 6 было создание цикла `for` для расчета вашего веса на Луне в течение 15 лет. Этот цикл можно оформить в виде функции. Создайте функцию, которая принимает начальный вес и величину, на которую вес увеличивается каждый год. Вызывать эту новую функцию нужно будет примерно так:

```
>>> moon_weight(30, 0.25)
```

Moon weight — лунный вес

#2. Функция лунного веса и количество лет

Измените функцию из предыдущего задания так, чтобы с ее помощью можно было рассчитывать вес для разного количества лет, например 5 или 20 лет. Пусть эта функция принимает три аргумента: начальный вес, прибавку веса в год и количество лет:

```
>>> moon_weight(90, 0.25, 5)
```

#3. Программа для лунного веса

Вместо простой функции, принимающей значения в виде аргументов, можно написать мини-программу, которая будет запрашивать эти значения с помощью `sys.stdin.readline()`. Тогда этой функции вообще не нужны аргументы:

```
>>> moon_weight()
```

Функция должна запросить начальный вес, потом прибавку веса в год и количество лет. Тогда работа с программой будет происходить примерно так:

```
Введите ваш нынешний земной вес
45
Введите ежегодный прирост вашего веса
0.4
```

```
Введите количество лет
12
```

Не забудьте импортировать модуль sys, прежде чем вводить код функции:

```
>>> import sys
```

8

КАК ПОЛЬЗОВАТЬСЯ КЛАССАМИ И ОБЪЕКТАМИ

Чем жираф похож на тротуар? Тем, что и жираф, и тротуар — *сущности*, которые в разговорном языке являются *именами существительными*, а в языке Python — *объектами*.

Концепция *объектов* имеет важное значение в мире программирования. Объекты — это способ организации кода в программе, а также способ разделения сложных задач на более простые, что облегчает их решение. (Кстати, в главе 4 нам уже доводилось использовать объект `Pen` (ручка) для рисования линий.)

Чтобы как следует разобраться, что такое объекты в Python, нужно поговорить об их типах. Начнем с жирафов и тротуаров.

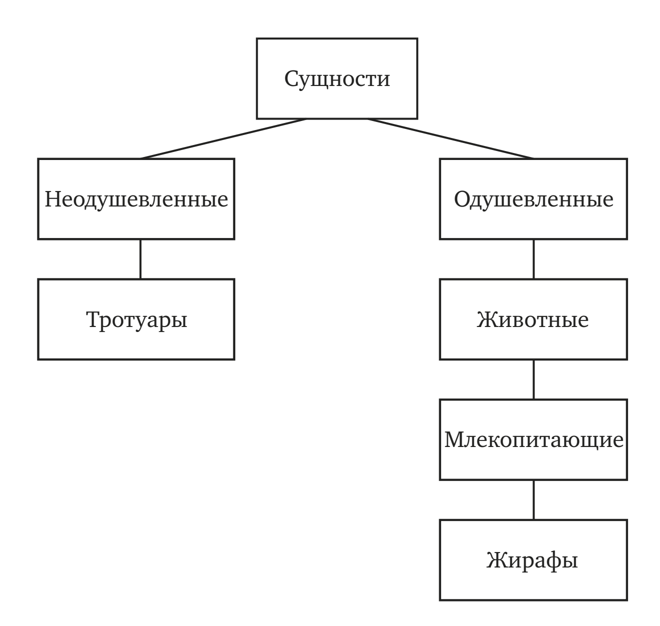
Жираф — это тип (или вид, как говорят биологи) млекопитающих, а млекопитающие, в свою очередь, являются одним из типов животных. Кроме того, жираф — одушевленный объект, поскольку он живой.

Теперь займемся тротуаром. Главное, что можно про него сказать, — он неживой, а значит, относится к неодушевленным объектам. Понятия «млекопитающее», «животное», «одушевленный» и «неодушевленный» — это способы классификации сущностей.



Разделяем сущности на классы

В языке Python объекты определяются *классами*, благодаря которым объекты можно разделять по смысловым группам. Вот диаграмма классов, которая соответствует нашим рассуждениям о жирафах и тротуарах:



Основной класс — это Сущности. Он имеет два подкласса — Неодушевленные и Одушевленные. Класс Неодушевленные содержит подкласс Тротуары, а класс Одушевленные — подкласс Животные, у которого есть подкласс Млекопитающие, а у него, в свою очередь, подкласс Жирафы.

Классы можно использовать и для организации фрагментов кода в Python-программе. Например, возьмем модуль `turtle`. Все действия, которые может выполнять этот модуль — перемещения черепашки вперед и назад, повороты направо и налево и так далее, — являются функциями класса `Pen`. Объект же является конкретной сущностью, принадлежащей этому классу. Для одного класса можно создать множество объектов, чем мы скоро и займемся.

Class — класс

Things — сущности, вещи

А пока давайте создадим такой же набор классов, как на нашей диаграмме, начиная сверху. Чтобы определить класс, нужно ввести ключевое слово `class`, а затем указать имя класса. Поскольку класс Сущности (по-английски — `Things`) — наиболее общий из всех, с него и начнем:

```
>>> class Things:
        pass
```

Мы дали классу имя `Things` и использовали конструкцию `pass`, обозначающую, что больше никакой информации мы указывать не будем. `Pass` — команда, с помощью которой можно создавать классы или функции, поначалу не программируя их поведение.

Дальше мы добавим остальные классы и зададим некоторые связи между ними.

Потомки и предки

Когда один класс является частным случаем (подклассом) другого класса, говорят, что первый класс — *потомок*, а второй — *предок*. Один и тот же класс может быть как потомком некоторых классов, так и предком для других классов. На нашей диаграмме класс, находящийся прямо над другим классом, — его предок, а класс под другим классом — его потомок. Например, классы Inanimate и Animate — потомки класса Things, который является их предком.

Inanimate — неодушевленные
Animate — одушевленные
Things — сущности

Для обозначения того, что создаваемый класс является потомком другого класса, нужно указать имя класса-предка в скобках после имени нового класса. Вот так:

```
>>> class Inanimate(Things):
        pass

>>> class Animate(Things):
        pass
```

Здесь мы создали класс Inanimate, указав, что его предком является класс Things, а затем создали класс Animate, также сделав Things его предком.

Теперь давайте создадим класс Sidewalks, указав его предком класс Inanimate:

Sidewalks — тротуары

```
>>> class Sidewalks(Inanimate):
        pass
```

И наконец, аналогичным образом создадим классы Animals, Mammals и Giraffes, не забыв указать в скобках имена классов-предков:

Animals — животные
Mammals — млекопитающие
Giraffes — жирафы

```
>>> class Animals(Animate):
        pass

>>> class Mammals(Animals):
        pass

>>> class Giraffes(Mammals):
        pass
```

Создаем объекты для классов

Теперь, когда у нас есть набор классов, пора создать принадлежащие им сущности. Предположим, у нас есть жираф по имени Реджинальд. Мы знаем, что он относится к классу Giraffes. Как именно описать в программе конкретного жирафа, которого зовут Реджинальд? Будем считать Реджинальда (reginald) *объектом* (или, как порой говорят, *экземпляром*) класса Giraffes. Чтобы «познакомить» Python с нашим Реджинальдом, нужно написать следующий код:

```
>>> reginald = Giraffes()
```

Этот код означает: создать объект класса Giraffes и присвоить его переменной reginald. После имени класса ставятся скобки, как после имени функции. Позже в этой главе мы выясним, как создавать объекты, используя указанные в этих скобках аргументы.

Что может делать объект reginald? Пока ничего. Чтобы объекты класса могли решать какие-то задачи, при создании этого класса нужно определить функции, с помощью которых объекты будут делать свое дело. Таким образом, вместо ключевого слова pass после определения класса мы можем описать принадлежащие ему функции.

Определение функций класса

В главе 7 мы уже говорили о функциях как о способе повторного использования кода. Определять функцию, которая принадлежит классу, следует так же, как и обычную функцию, но после определения класса и с отступом. Например, вот обычная функция, не имеющая никакого отношения к классам:

This is a normal function — это нормальная функция

```
>>> def this_is_a_normal_function():
        print('Я — обычная функция')
```

А вот несколько функций, принадлежащих классу:

This is my silly class — это мой глупый класс

This is a class function — это функция класса

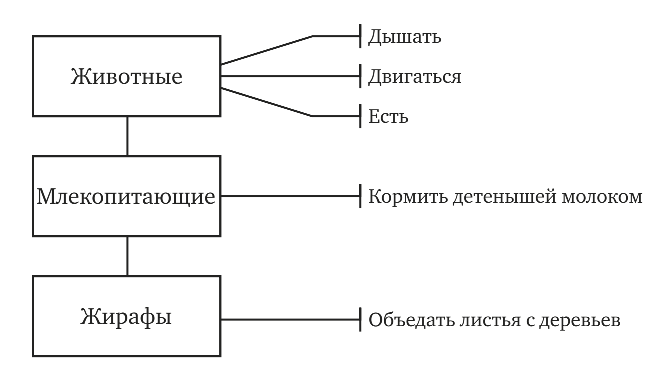
This is also a class function — это тоже функция класса

```
>>> class ThisIsMySillyClass:
        def this_is_a_class_function():
            print('Я - функция класса')
        def this_is_also_a_class_function():
            print('Я тоже функция класса, понятно?')
```

Используем функции для задания характеристик класса

Рассмотрим классы-потомки класса `Animate`, которым мы дали определение на стр. 97. Можно задать каждому из них *характеристики* (описания, что это за класс и что он может делать). Характеристики — это особенности, присущие всем объектам данного класса, а также объектам классов-потомков.

Например, что общего у всех животных? Они дышат, двигаются и едят. А у млекопитающих? Они вскармливают своих детенышей молоком, а также дышат, двигаются и едят. Жирафы, как известно, объедают листья с верхушек деревьев и, как прочие млекопитающие, кормят детенышей молоком, дышат, двигаются и едят. Если добавить эти характеристики к нашей диаграмме, получится вот что:



Данные характеристики можно представить как действия, или *функции* — то, что объект или класс может делать.

Для добавления функций класса мы используем ключевое слово `def`, поэтому класс `Animals` будет выглядеть примерно так:

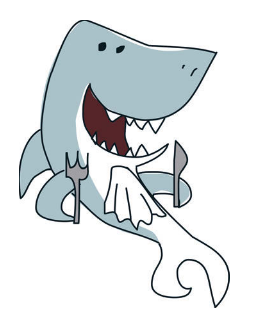
```
>>> class Animals(Animate):
        def breathe(self):
            pass
        def move(self):
            pass
        def eat_food(self):
            pass
```

Breathe — дышать

Move — двигаться

Eat food — есть еду

В первой строке мы определили класс — так, как делали это раньше. Однако следующей строкой вместо `pass` дали определение функции `breathe` с одним аргументом `self`. Аргумент `self` дает функции класса возможность вызывать другие функции этого класса (а также классов-предков). Об использовании этого аргумента мы поговорим немного позже.



Ключевое слово pass на следующей строке говорит о том, что мы не собираемся больше ничего сообщать о функции breathe — она пока не выполняет никаких действий. Затем идут определения функций move и eat_food, которые тоже пока ничего не делают. Скоро мы перепишем наши классы и добавим в функции подобающий код. Это довольно типичный подход к разработке программ. Часто программисты создают классы с функциями, которые ничего не делают, чтобы лучше представить, какими должны быть эти классы, прежде чем переходить к реализации отдельных функций.

Feed young with milk — кормить детенышей молоком
Eat leaves from trees — есть листья деревьев

Давайте создадим функции и для двух других классов, а именно функцию feed_young_with_milk для класса Mammals и функцию eat_leaves_from_trees для класса Giraffes. Каждый класс будет иметь доступ к характеристикам (то есть к функциям) своего предка. Следовательно, ни к чему описывать все особенности класса в нем самом, перегружая его сложным кодом: функции могут располагаться в классах-предках, к которым они логически относятся. Таким образом можно создавать простые и легко читаемые классы.

```
>>> class Mammals(Animals):
        def feed_young_with_milk(self):
            pass

>>> class Giraffes(Mammals):
        def eat_leaves_from_trees(self):
            pass
```

Зачем нужны классы и объекты?

Мы снабдили классы функциями, однако зачем нужны классы и объекты, если можно было написать обычные функции с именами breathe, move, eat_food и так далее?

Прояснить этот вопрос нам поможет жираф Реджинальд, которого мы создали как объект класса Giraffes:

```
>>> reginald = Giraffes()
```

Поскольку reginald является объектом, мы можем вызывать функции, определенные в его классе (Giraffes) и в его классах-предках. Чтобы вызвать такую функцию для объекта, нужно после имени объекта ввести точку, а затем имя функции. Соответственно, можно дать

Реджинальду команду двигаться или есть, вызывая функции таким образом:

```
>>> reginald = Giraffes()
>>> reginald.move()
>>> reginald.eat_leaves_from_trees()
```

Теперь предположим, что у Реджинальда есть друг — жираф, которого зовут Гарольд. Давайте создадим еще один объект класса `Giraffes` с именем `harold`:

```
>>> harold = Giraffes()
```

Поскольку мы используем объекты и классы, при вызове функции `move` можно указать, к какому жирафу этот вызов относится. Например, если мы хотим, чтобы Гарольд передвинулся, а Реджинальд остался на месте, нужно вызвать `move` для объекта `harold`. Вот так:

```
>>> harold.move()
```

И тогда двигаться будет только Гарольд.

Чтобы это увидеть, слегка изменим код наших классов — вместо `pass` используем в теле каждой функции команду `print`:

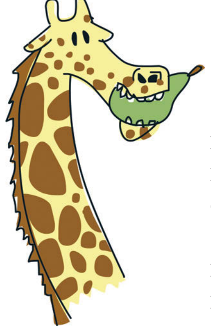
```
>>> class Animals(Animate):
        def breathe(self):
            print('дышит')
        def move(self):
            print('двигается')
        def eat_food(self):
            print('ест')

>>> class Mammals(Animals):
        def feed_young_with_milk(self):
            print('кормит детенышей молоком')

>>> class Giraffes(Mammals):
        def eat_leaves_from_trees(self):
            print('ест листья')
```

Теперь, когда мы создадим объекты `reginald` и `harold` и станем вызывать для них функции, будет видно, что происходит:

```
>>> reginald = Giraffes()
>>> harold = Giraffes()
>>> reginald.move()
двигается
>>> harold.eat_leaves_from_trees()
ест листья
```



В первых двух строках кода мы создали переменные reginald и harold, которые являются объектами класса Giraffes. Затем вызываем для reginald функцию move, и в следующей строке Python пишет «двигается». Аналогичным образом вызываем функцию eat_leaves_from_trees для harold, и Python пишет «ест листья». Если бы это были настоящие жирафы, а не объекты Python, один из них двигался бы, а другой ел.

Объекты и классы в картинках

Как насчет более наглядного, графического подхода к объектам и классам?

Давайте вернемся к модулю turtle, с которым мы упражнялись в главе 4. При вызове функции turtle.Pen() Python создает для нас объект класса Pen из модуля turtle (аналогично тому, как мы создавали объекты reginald и harold класса Giraffe в предыдущем разделе). Так же, как мы создавали двух жирафов, можно создать двух черепашек. Назовем их Эвери (Avery) и Кейт (Kate):

```
>>> import turtle
>>> avery = turtle.Pen()
>>> kate = turtle.Pen()
```

Каждый из объектов-черепашек (avery и kate) принадлежит к классу Pen.

Теперь объекты покажут нам свою мощь, ведь после того как объекты-черепашки созданы, можно вызывать функции каждого объекта по отдельности, чтобы черепашки передвигались и рисовали независимо друг от друга.

```
>>> avery.forward(50)
>>> avery.right(90)
>>> avery.forward(20)
```

Эта последовательность команд переместит Эвери на 50 пикселей вперед, повернет на 90 градусов вправо и переместит вперед на 20 пикселей.

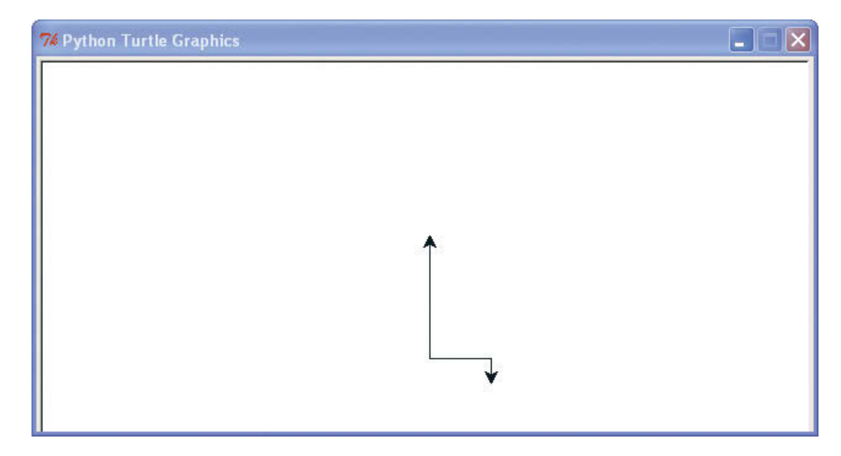
В итоге черепашка будет смотреть вниз. Помните, что вначале черепашки всегда развернуты вправо.

Теперь настала очередь Кейт.

```
>>> kate.left(90)
>>> kate.forward(100)
```

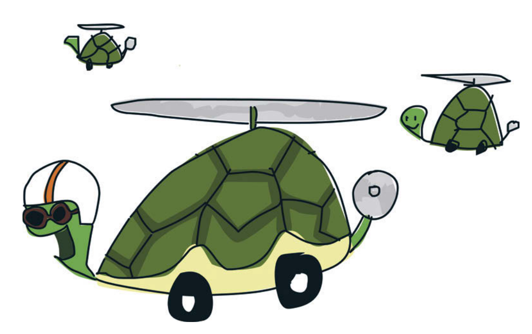
Мы развернули Кейт на 90 градусов влево и передвинули вперед на 100 пикселей. Она будет смотреть вверх.

У нас получилась линия с двумя стрелочками, направленными в разные стороны, причем каждая стрелочка соответствует одному из объектов-черепашек: Эвери смотрит вниз, а Кейт — вверх.

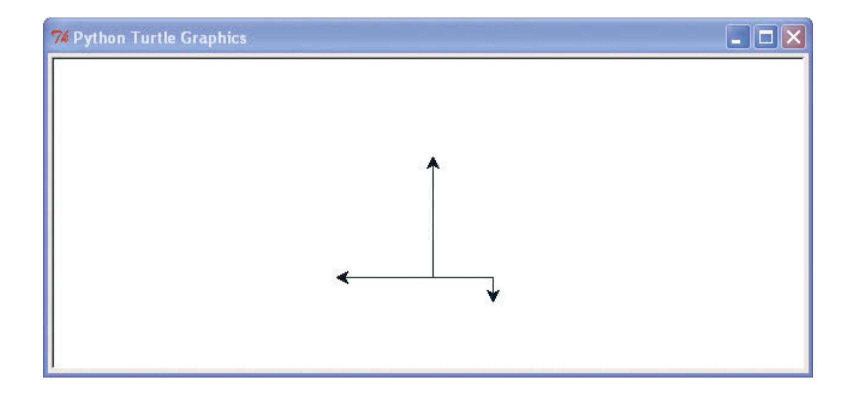


Теперь добавим еще одну черепашку — Джейкоба (Jacob) и переместим его, не беспокоя Эвери и Кейт.

```
>>> jacob = turtle.Pen()
>>> jacob.left(180)
>>> jacob.forward(80)
```



Создаем новый объект класса `Pen` с именем `jacob`, разворачиваем его на 180 градусов и перемещаем вперед на 80 пикселей. Теперь у нас три черепашки, а картинка выглядит так:



Не забывайте, что каждый раз при создании черепашки с помощью `turtle.Pen()` в программе возникает новый самостоятельный объект. Каждый из таких объектов принадлежит к классу `Pen`, и для каждого из них мы можем вызывать одни и те же функции. Однако действия одной черепашки не связаны с действиями остальных: подобно нашим жирафам (Реджинальду и Гарольду), черепашки Эвери, Кейт и Джейкоб — независимые объекты. Также обратите внимание: если мы создадим новый объект, сохранив его в переменной, где до этого был другой объект, это не означает, что прежний объект тут же исчезнет. Попробуйте сами создать еще одну черепашку Кейт и подвигать ее по холсту.

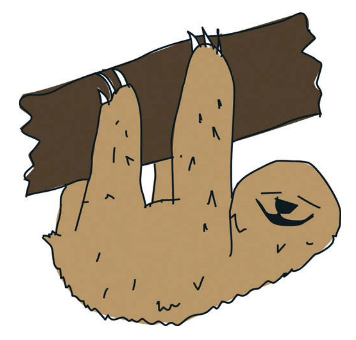
Другие полезные свойства объектов и классов

Классы и объекты позволяют удобно группировать функции. Также они помогают представлять программу как набор небольших фрагментов кода.

Представьте действительно большую компьютерную программу — видеоигру или текстовый процессор. Если воспринимать такую программу как неделимое целое, очень сложно будет понять, как она работает, поскольку объем кода слишком велик. Но стоит разбить эту гигантскую программу на части — и вы разберетесь, что делает каждая из частей (разумеется, если знаете язык, на котором программа написана).

Кроме того, если большую программу создавать по частям, становится проще распределять работу между разными программистами. Над наиболее сложными программами (такими как веб-браузер, например) трудятся целые команды программистов по всему миру, одновременно работая над разными частями кода.

Допустим, вам нужно доработать некоторые из созданных в этой главе классов (Animals, Mammals и Giraffes), но времени не хватает, и вы обратились за помощью к друзьям. Тогда работу несложно поделить на части: один человек займется классом Animals, другой — классом Mammals, а кому-то достанется Giraffes.

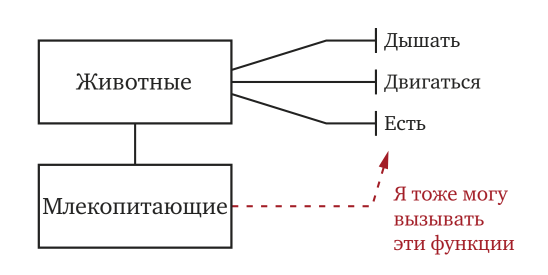


Унаследованные функции

Наверное, вы уже поняли (если читали внимательно), что программисту, который будет дорабатывать класс Giraffes, крупно повезло, ведь в этом классе можно пользоваться функциями классов Animals и Mammals, которые написаны другими людьми. Класс Giraffes *наследует* функции класса Mammals, который, в свою очередь, наследует функции класса Animals. Иными словами, создав объект-жирафа, мы получим доступ к функциям класса Giraffes, а также классов Mammals и Animals. Аналогично, создав объект-млекопитающее, мы сможем пользоваться функциями классов Mammals и Animals.

Mammals — млекопитающие

Посмотрите еще раз на связь между классами Animals, Mammals и Giraffes. Класс Animals является предком класса Mammals, а он, в свою очередь, — предком класса Giraffes.



Хоть reginald — объект класса Giraffes, для него можно вызвать функцию move (двигаться), определенную в классе Animals, так как функции, определенные в классах-предках, доступны и классам-потомкам:

```
>>> reginald = Giraffes()
>>> reginald.move()
двигается
```

Действительно, все функции, которые мы определили для классов Animals и Mammals, можно вызвать для объекта reginald, поскольку он наследует эти функции:

```
>>> reginald = Giraffes()
>>> reginald.breathe()
дышит
>>> reginald.eat_food()
ест
>>> reginald.feed_young_with_milk()
кормит детенышей молоком
```

Функции, вызывающие другие функции

Вызывая функцию объекта, мы используем имя переменной, в которой этот объект хранится. Например, так мы вызывали для жирафа Реджинальда функцию move:

```
>>> reginald.move()
```

Если же требуется вызвать функцию move из другой функции, определенной в классе Giraffes, вместо имени переменной следует использовать аргумент self. Таким способом можно вызвать любую функцию класса (или классов-предков) из другой функции класса. Например, добавим в класс Giraffes функцию find_food:

Find food — найти еду

```
>>> class Giraffes(Mammals):
        def find_food(self):
            self.move()
            print("Я нашел еду!")
            self.eat_food()
```

Мы создали функцию, которая вызывает две другие функции. Это весьма распространенный прием программирования — зачастую, создав полезную функцию, удобно вызывать ее из еще одной функции (и мы займемся этим в главе 13 при создании игры, для которой нам понадобятся более сложные функции).

Добавим в класс Giraffes еще несколько функций, использующих self:

```
>>> class Giraffes(Mammals):
        def find_food(self):
            self.move()
            print("Я нашел еду!")
            self.eat_food()
        def eat_leaves_from_trees(self):
            self.eat_food()
        def dance_a_jig(self):
            self.move()
            self.move()
            self.move()
            self.move()
```

Dance a jig — танцевать джигу

При создании функций eat_leaves_from_trees и dance_a_jig мы использовали функции eat_food и move, принадлежащие

классу-предку Animals. Это допустимо, поскольку функции наследуются. Таким образом можно создавать (а затем вызывать для конкретных объектов класса) функции, которые выполняют несколько действий подряд. Смотрите, что произойдет при вызове dance_a_jig, — наш жираф передвинется 4 раза (сообщение «двигается» будет напечатано четырежды):

```
>>> reginald = Giraffes()
>>> reginald.dance_a_jig()
двигается
двигается
двигается
двигается
```

Инициализация объектов

При создании объекта иногда нужно задать какие-нибудь значения (их также называют *свойствами*), которые понадобятся для работы с этим объектом. Подготовка объекта к использованию называется его *инициализацией*.

Например, мы хотим задавать количество пятен на шкуре жирафа при создании каждого объекта класса Giraffes (то есть во время инициализации объекта). Для этого нужно создать функцию с именем __init__ (обратите внимание, что в начале и в конце стоят по два знака подчеркивания). Функция с таким названием имеет особое значение для классов — позволяет задать свойства объекта при его создании, поскольку для каждого нового объекта она вызывается автоматически. Вот как пользоваться этой функцией:

```
>>> class Giraffes:
        def __init__(self, spots):
            self.giraffe_spots = spots
```

Введя def __init__(self, spots):, мы создали функцию инициализации, которая принимает два аргумента — self и spots. Как и остальные функции классов, функция инициализации должна принимать значение self первым аргументом. В теле функции строка self.giraffe_spots = spots присваивает значение аргумента spots свойству объекта giraffe_spots (свойство — это принадлежащая объекту переменная). Таким образом, эта строка означает «Взять значение аргумента spots и поместить его в переменную объекта (свойство объекта) с именем giraffe_spots». Как и в случае вызова одной функции класса из другой, для доступа к свойствам используется аргумент self.

Init — от initialize — установить начальное состояние
Spots — пятна

Теперь испытаем функцию инициализации, создав еще пару объектов — тоже жирафов (назовем их Освальд и Гертруда) — и проверив, сколько у них пятен:

```
>>> ozwald = Giraffes(100)
>>> gertrude = Giraffes(150)
>>> print(ozwald.giraffe_spots)
100
>>> print(gertrude.giraffe_spots)
150
```

Мы создали объект класса `Giraffes`, указав в скобках значение 100. В результате была вызвана функция `__init__` с аргументом `spots`, равным 100. Затем мы создали еще один объект класса `Giraffes`, на этот раз указав для аргумента `spots` значение 150. И наконец, напечатав значения свойств `giraffe_spots` для обоих объектов, мы получили 100 и 150. Ура, работает!

Запомните, что после создания объекта (такого как объект `ozwald` в этом примере) для обращения к его функциям и свойствам используются точка и имя переменной (например, `ozwald.giraffe_spots`). Однако при написании функций класса обращаться к этим же свойствам и функциям нужно через аргумент `self` (`self.giraffe_spots`).

Что мы узнали

В этой главе мы использовали классы для классификации сущностей, а также создавали объекты (экземпляры) этих классов. Узнали, что класс-потомок наследует функции классов-предков и, даже если объекты принадлежат одному классу, они могут отличаться друг от друга (например, у разных объектов — жирафов — может быть разное количество пятен). Мы научились вызывать для объекта функции, определенные в его классе, и сохранять значения в свойствах объектов. И наконец, мы выяснили, как с помощью аргумента `self` получить доступ к функциям и свойствам класса из других его функций. Все это — фундаментальные понятия языка Python, и вы еще не раз встретитесь с ними на страницах этой книги.

Упражнения

Чтобы как следует разобраться в классах и объектах, нужно начать ими пользоваться. Постарайтесь выполнить эти задания, не подглядывая в подсказки.

#1. Жирафий танец

Добавьте в класс `Giraffes` функции, при вызове которых жираф переставлял бы правую или левую ногу вперед либо назад. Назвать их можно так: `left_foot_forward`, `left_foot_back`, `right_foot_forward` и `right_foot_back`. Функция, которая ставит левую ногу жирафа вперед, может выглядеть примерно так:

```
>>> def left_foot_forward(self):
            print('левая нога впереди')
```

Left foot forward — левая нога вперед

Left foot back — левая нога назад

Right foot forward — правая нога вперед

Right foot back — правая нога назад

Dance — танцевать

Теперь создайте функцию `dance`, которая научит Реджинальда танцевать (вызывая четыре только что созданные функции для передвижения ног). В результате должен получиться несложный танец:

```
>>> reginald = Giraffes()
>>> reginald.dance()
левая нога впереди
левая нога сзади
правая нога впереди
правая нога сзади
левая нога сзади
правая нога сзади
правая нога впереди
левая нога впереди
```

#2. Черепашьи вилы

При помощи четырех черепашек (объектов класса `Pen`) изобразите вилы, как на этой картинке (длину линий выберите на свой вкус). Не забудьте первым делом импортировать модуль `turtle`!



Встроенные функции Python

В комплекте с языком Python идет богатый набор программных инструментов, включая множество готовых к использованию функций и модулей. Эти инструменты, то есть фрагменты кода, могут значительно облегчить работу над созданием программ.

Как вы уже знаете из главы 7, прежде чем использовать модуль, его нужно импортировать. Однако *встроенные функции* языка Python импортировать не нужно, ими можно пользоваться сразу после запуска оболочки. В этой главе мы познакомимся с некоторыми из наиболее полезных встроенных функций и отдельно остановимся на функции `open`, которая позволяет открывать файлы для чтения и записи данных.

Использование встроенных функций

Рассмотрим 12 функций, которые часто используются в Python-программах: что они делают, как их вызывать и для чего они могут понадобиться в вашем коде.

Функция abs

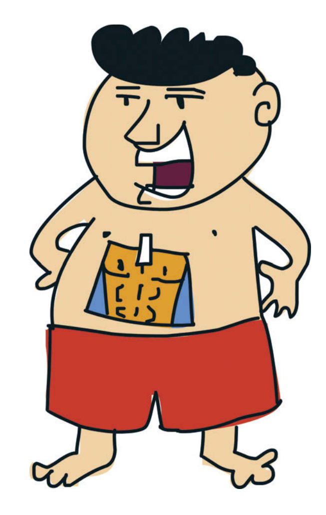
Функция `abs` возвращает *абсолютное значение* (*модуль*) числа, то есть само число без знака. Например, абсолютное значение 10 равно 10, а абсолютное значение –10 тоже равно 10.

Abs — от absolute — абсолютный

При вызове функции `abs` нужно передать число или переменную таким образом:

```
>>> print(abs(10))
10
>>> print(abs(-10))
10
```

Эту функцию можно использовать, к примеру, для получения абсолютной величины перемещения персонажа в игре независимо от направления его движения. Скажем, персонаж сделал три шага вправо (положительное значение 3), а затем 10 шагов влево (отрицательное значение –10). Если не обращать внимания на направление, абсолютные значения этих перемещений будут равны 3 и 10. Этим можно воспользоваться в настольной игре, где вы бросаете пару кубиков, а потом делаете необходимое количество ходов (которые определяются суммой очков, выпавших на обоих кубиках) в любом направлении. Если поместить количество ходов в переменную, с помощью приведенного ниже кода можно узнать, будет персонаж двигаться или нет. Предположим, когда игрок сделает выбор, мы хотим отобразить сообщение (в данном случае просто напечатать «Персонаж двигается»):



```
>>> steps = -3
>>> if abs(steps) > 0:
        print('Персонаж двигается')
```

Steps — шаги

Если бы мы не пользовались `abs`, конструкция `if` выглядела бы примерно так:

```
>>> steps = -3
>>> if steps < 0 or steps > 0:
        print('Персонаж двигается')
```

Как видите, применение `abs` делает выражение после `if` немного короче и понятнее.

Функция bool

Название функции `bool` — это сокращение от термина *boolean*, то есть «*булево значение*». Так программисты называют данные, у которых может быть лишь одно из двух значений: `True` или `False`.

Bool — от boolean — булево значение

True — истина

False — ложь

Функция bool принимает один аргумент и в зависимости от его значения возвращает True или False. Если аргумент числовой, функция вернет False для значения 0 и True во всех остальных случаях. Вот примеры использования bool с разными числами:

```
>>> print(bool(0))
False
>>> print(bool(1))
True
>>> print(bool(1123.23))
True
>>> print(bool(-500))
True
```

None — отсутствует

При использовании функции bool с другими типами данных, например со строками, она вернет False для пустого значения (иначе говоря, для значения None или пустой строки). В остальных случаях она вернет True:

```
>>> print(bool(None))
False
>>> print(bool('a'))
True
>>> print(bool(' '))
True
>>> print(bool('Как назвать свинью на соревнованиях по карате? ↵
Свиная отбивная!'))
True
```

Также функция bool вернет False для списков, кортежей или словарей, которые не содержат значений (и True в противном случае):

My silly list — мой глупый список

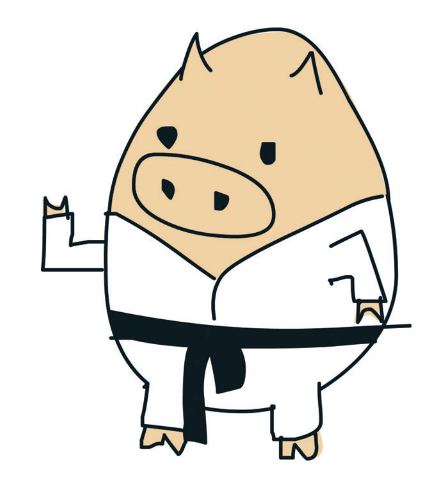
```
>>> my_silly_list = []
>>> print(bool(my_silly_list))
False
>>> my_silly_list = ['s', 'i', 'l', 'l', 'y']
>>> print(bool(my_silly_list))
True
```

Функция bool пригодится, если нужно определить, задано значение или нет. Например, если мы запросим у пользователя программы год рождения, можно использовать if совместно с bool для проверки введенного значения:

```
>>> year = input('Год рождения: ')
Год рождения:
>>> if not bool(year.rstrip()):
        print('Вам следует ввести год своего рождения')
Вам следует ввести год своего рождения
```

Year — год

В первой строке кода мы с помощью функции input запрашиваем ввод значения с клавиатуры, сохраняя его в переменной year. Если в ответ нажать только клавишу **Enter** (не вводя перед этим ничего), в переменной year окажется пустая строка. (В главе 7 мы пользовались другим способом ввода с клавиатуры — sys.stdin.readline().)



В следующей строке кода мы с помощью if проверяем значение переменной, которое перед этим обрабатывается функцией rstrip (она удаляет все пробелы в конце строки, если пользователь их введет). В нашем примере пользователь не ввел значение, поэтому функция bool вернет False. Поскольку в конструкции if здесь используется операция not (не), что означает «если функция bool *не* вернет True», на экране появится сообщение «Вам следует ввести год своего рождения».

Функция dir

Функция dir выдает информацию о любом переданном ей значении. В сущности, она сообщает, какие функции можно использовать с этим значением, перечисляя их имена в алфавитном порядке.

Например, чтобы получить имена функций, которые можно использовать со списком, введите:

```
>>> dir(['это', 'короткий', 'список'])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Функция dir работает практически с любыми типами значений, включая строки, числа, функции, модули, объекты и классы. Однако

порой информация, которую функция возвращает, не слишком полезна. Например, вызвав dir с числом 1 в качестве аргумента, вы увидите множество специальных функций (их имена начинаются и заканчиваются знаками подчеркивания), которые редко используются в обычных программах, и вам они вряд ли пригодятся:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__divmod__', '__doc__', '__eq__',
'__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__',
'__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'imag', 'numerator', 'real']
```

Popcorn — попкорн

Используйте функцию dir, чтобы оперативно узнать, как быть с той или иной переменной. Например, передав dir переменную popcorn, содержащую строковое значение, вы увидите набор функций строкового класса str (все строки в Python являются объектами строкового класса):

```
>>> popcorn = 'Я люблю попкорн!'
>>> dir(popcorn)
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

Теперь можно воспользоваться функцией help и получить краткую справку по любой функции из этого списка. Например, справка для функции upper:

Help — помощь

Upper — здесь «заглавный регистр символов»

```
>>> help(popcorn.upper)
Help on built-in function upper:

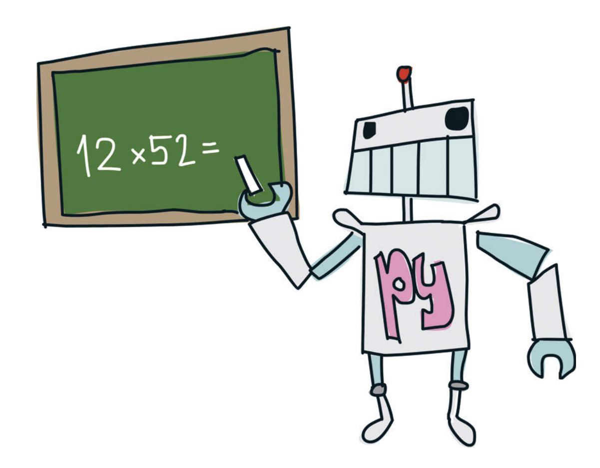
upper(...)
    S.upper() -> str
    Return a copy of S converted to uppercase.
```

Здесь многоточие (...) означает, что upper — функция строкового класса, которая не принимает аргументы. Стрелочка (->) на следующей строке показывает, что функция возвращает строку (str). И наконец, далее вкратце описано, что делает эта функция (возвращает строку в верхнем регистре, т. е. все буквы — прописные).

Функция eval

Функция eval принимает в качестве аргумента строку и выполняет ее, считая, что это код на языке Python. Например, в результате вызова eval('print("вау!")') будет выполнена команда print("вау!").

Eval — от evaluate — вычислить, выполнить



Функция eval работает только с простыми выражениями вроде такого:

```
>>> eval('10*5')
50
```

Выражения, состоящие из нескольких строк кода (например, конструкции if), выполнить, как правило, не удастся:

```
>>> eval('''if True:
        print("это не сработает")''')
Traceback (most recent call last):
  File "<pyshell#35>", line 2, in <module>
    print("this won't work at all")''')
  File "<string>", line 1
    if True:
     ^
SyntaxError: invalid syntax
```

Функция eval часто используется, чтобы обработать ввод пользователя как Python-выражение. Например, можно написать простую программу-калькулятор, которая запрашивает математическое выражение и с помощью eval вычисляет результат.

Введенное пользователем выражение считывается как строка, и для его вычисления нужно преобразовать строку в числа и операторы. Функция eval сделает это автоматически:

```
>>> your_calculation = input('Введите выражение: ')
Введите выражение: 12*52
>>> eval(your_calculation)
624
```

Your calculation — ваш расчет

Мы использовали input для считывания введенной пользователем строки и ее сохранения в переменной your_calculation. В качестве выражения мы ввели 12*52 (предположим, это ваш возраст, умноженный на количество недель в году). Затем вызвали функцию eval, которая вычислила результат: 624.

Функция exec

Exec — от execute — выполнять

Функция exec похожа на eval, но с ее помощью можно выполнять более сложный код. Кроме того, exec, в отличие от eval, не возвращает значения. Например:

My small program — моя маленькая программа

```
>>> my_small_program = '''print('бутерброд')
print('с колбасой')'''
>>> exec(my_small_program)
бутерброд
с колбасой
```

Мы сначала создали переменную my_small_program, поместив туда строку с двумя командами print, разделенными символом новой строки, а затем вызвали функцию exec для выполнения кода, хранящегося в my_small_program.

Функция exec может пригодиться для запуска небольших программ, которые главная Python-программа считывает из файлов, то есть для выполнения программ внутри другой программы. Это нужно при разработке больших и сложных приложений. Скажем, можно написать игру «Дуэль роботов», в которой два робота перемещаются по экрану, стараясь атаковать друг друга. При этом каждый игрок вместо того, чтобы управлять своим роботом напрямую, создает для него короткую Python-программу. Игра могла бы считывать такие программки и выполнять их с помощью exec.

Функция float



Функция float преобразует строку или число в *вещественное число*, то есть в формат с десятичной запятой. Например, число 10 — это *целое число*, однако 10.0, 10.1 и 10.253 — вещественные числа (не забывайте, что в Python при записи десятичных дробей используется точка, а не запятая). В отличие от целых, вещественные числа подходят для вычислений, связанных с деньгами. Также они используются в графических программах (в частности, в трехмерных играх) для определения, как и где рисовать изображения на экране.

Преобразовать строку в вещественное число можно так:

```
>>> float('12')
12.0
```

В строке может быть и десятичная точка:

```
>>> float('123.456789')
123.456789
```

Функция float пригодится для преобразования введенных значений в числа подходящего формата. В особенности это важно, если требуется сравнить значение, заданное пользователем, с другими значениями. Например, с помощью следующего кода можно проверить, что возраст пользователя не превышает 13 лет:

```
>>> your_age = input('Введите ваш возраст: ')
Введите ваш возраст: 20
>>> age = float(your_age)
>>> if age > 13:
        print('Вы на %s лет старше, чем положено' % (age - 13))
Вы на 7.0 лет старше, чем положено
```

Функция int

Функция `int` служит для преобразования строки или числа в целое число, при этом знаки после запятой (если они есть) отбрасываются. Преобразовать вещественное число в целое можно так:

```
>>> int(123.456)
123
```

А так — преобразовать в целое число строку:

```
>>> int('123')
123
```

Однако при попытке воспользоваться функцией `int` для преобразования строки с вещественным числом в целое вы получите ошибку:

```
>>> int('123.456')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    int('123.456')
ValueError: invalid literal for int() with base 10: '123.456'
```

Как видите, Python вывел сообщение об ошибке — `ValueError`.

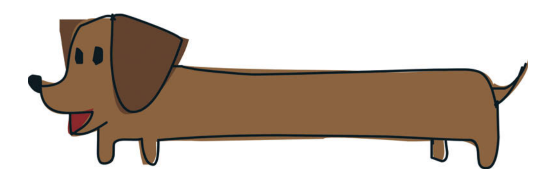
Функция len

Len — от length — длина

Функция `len` возвращает «длину» объекта, то есть количество элементов в нем. Если это строка, функция вернет количество символов. Например, так можно узнать длину строки «это тестовая строка»:

```
>>> len('это тестовая строка')
19
```

Если передать в функцию len список или кортеж, результатом будет количество элементов в этом списке или кортеже:



```
>>> creature_list = ['единорог', 'циклоп', 'фея', 'эльф', 'дракон',
                     'тролль']
>>> print(len(creature_list))
6
```

Creature list — список существ

Для словаря len также вернет количество элементов:

```
>>> enemies_map = {'Бэтмен' : 'Джокер',
                   'Супермен' : 'Лекс Лютор',
                   'Человек-паук' : 'Зеленый гоблин'}
>>> print(len(enemies_map))
3
```

Enemies map — карта врагов

Особенно полезна функция len при работе с циклами. Например, с ее помощью можно напечатать индексы всех элементов в списке:

```
   >>> fruit = ['яблоко', 'банан', 'клементин', 'питайя']
❶ >>> length = len(fruit)
❷ >>> for x in range(0, length):
❸         print('фрукт с индексом %s: %s' % (x, fruit[x]))

фрукт с индексом 0: яблоко
фрукт с индексом 1: банан
фрукт с индексом 2: клементин
фрукт с индексом 3: питайя
```

Fruit — фрукт

В строке с меткой ❶ мы сохранили длину списка в переменной length, затем в строке ❷ использовали это значение как аргумент функции range при создании цикла. В строке ❸ в цикле перебираем элементы списка, печатая индекс каждого элемента и его значение. Также с помощью функции len можно напечатать каждый второй или каждый третий элемент из списка строк.

Функции max и min

Функция max возвращает наибольший элемент списка, кортежа или строки. Например, для списка чисел:

Max — от maximum — наибольший

```
>>> numbers = [5, 4, 10, 30, 22]
>>> print(max(numbers))
30
```

Можно также найти наибольший символ в строке:

```
>>> strings = 'строкаСТРОКА'
>>> print(max(strings))
т
```

Как видно из этого примера, буквы сравниваются по их позициям в алфавите — чем ближе буква к концу алфавита, тем она больше, при этом строчные буквы всегда больше прописных, из-за чего наибольший символ здесь «т», а не «Т».

Однако не обязательно передавать в функцию max переменную, где хранится список, кортеж или словарь. Можно просто указать значения через запятую:

```
>>> print(max(10, 300, 450, 50, 90))
450
```

Min — от minimum — наименьший

Функция min похожа на max, только возвращает она наименьший элемент списка, кортежа или строки. Вот тот же пример со списком чисел, где вместо max используется min:

```
>>> numbers = [5, 4, 10, 30, 22]
>>> print(min(numbers))
4
```

Представьте, что вы играете в «Угадай число» против команды из четырех игроков, каждый из которых должен назвать число, меньше загаданного вами. Если хотя бы один из ваших оппонентов назовет число больше, все четверо проиграют, но если все назовут числа меньше, проиграете вы. С помощью max можно быстро проверить, меньше ли все варианты ваших оппонентов загаданного вами числа:

Guess this number — угадать число

Player guesses — угадывает игрок

```
>>> guess_this_number = 61
>>> player_guesses = [12, 15, 70, 45]
>>> if max(player_guesses) > guess_this_number:
        print('Бабах! Вы проиграли')
```

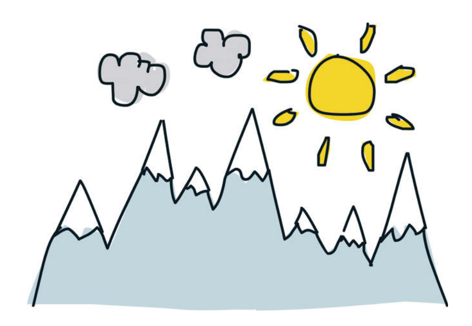
```
else:
        print('Вы победили')

Бабах! Вы проиграли
```

Число, которое загадали вы, хранится в переменной guess_this_number, а варианты ваших противников — в списке player_guesses. Конструкция if сравнивает максимальное из названных вашими противниками чисел с guess_this_number, и если хоть один вариант окажется больше вашего, будет напечатано сообщение «Бабах! Вы проиграли».

Функция range

Как вы уже знаете, функция range используется в основном в циклах, чтобы выполнить блок кода определенное количество раз. Первые два аргумента range называются *старт* и *стоп*. Пример использования range с этими двумя аргументами я уже приводил, когда рассказывал об использовании len с циклами.



Последовательность чисел, которую генерирует range, начинается со значения, переданного в первом аргументе, и заканчивается числом, которое на единицу меньше второго аргумента. Например, вот что мы увидим, напечатав числа, сгенерированные функцией range с аргументами 0 и 5:

```
>>> for x in range(0, 5):
        print(x)

0
1
2
3
4
```

На самом деле range возвращает специальный объект, называемый *итератором*, который несколько раз повторяет некое действие. В данном случае он при каждом обращении выдает очередное число из последовательности.

Итератор можно преобразовать в список (с помощью функции list). Напечатав такой список, вы увидите все значения, которые вернул итератор:

List — список

```
>>> print(list(range(0, 5)))
[0, 1, 2, 3, 4]
```

Функция range может принимать и третий аргумент, называемый *шагом* (если этот аргумент не указан, шаг считается равным 1). Что же будет, если передать в качестве шага число 2? А вот что:

Count by twos — считать двойками

```
>>> count_by_twos = list(range(0, 30, 2))
>>> print(count_by_twos)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Как видите, каждый последующий элемент списка больше предыдущего на 2. И заканчивается список числом 28, которое на 2 меньше стоп-значения — числа 30. Можно указать и отрицательный шаг:

Count down by twos — считать двойками в обратную сторону

```
>>> count_down_by_twos = list(range(40, 10, -2))
>>> print(count_down_by_twos)
[40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12]
```

Функция sum

Sum — сумма, складывать

Функция sum складывает элементы списка, возвращая их сумму. Пример:

My list of numbers — мой список чисел

```
>>> my_list_of_numbers = list(range(0, 500, 50))
>>> print(my_list_of_numbers)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]
>>> print(sum(my_list_of_numbers))
2250
```

В первой строке мы создали список чисел в диапазоне от 0 до 500, используя range с шагом 50, и вывели этот список на экран. Затем мы передали список функции sum и напечатали результат — 2250.

Работа с файлами

Файлы в Python-программах такие же, как и другие файлы на вашем компьютере: документы, изображения, музыка, игры и так далее. Вся информация на компьютере хранится в файлах.

Open — открыть

Давайте посмотрим, как с помощью встроенной функции open открывать файлы и работать с ними. Сначала нужно создать для наших экспериментов новый файл.

Создание файла

Мы будем экспериментировать с текстовым файлом под названием *test.txt*. Выполните перечисленные далее шаги для той операционной системы, которую вы используете.

Создание нового файла в Windows

Выполните следующие шаги для создания файла *test.txt*:

1. Выберите **Start ▸ All Programs ▸ Accessories ▸ Notepad** (**Пуск ▸ Все программы ▸ Стандартные ▸ Блокнот**).

2. Введите несколько строк текста.

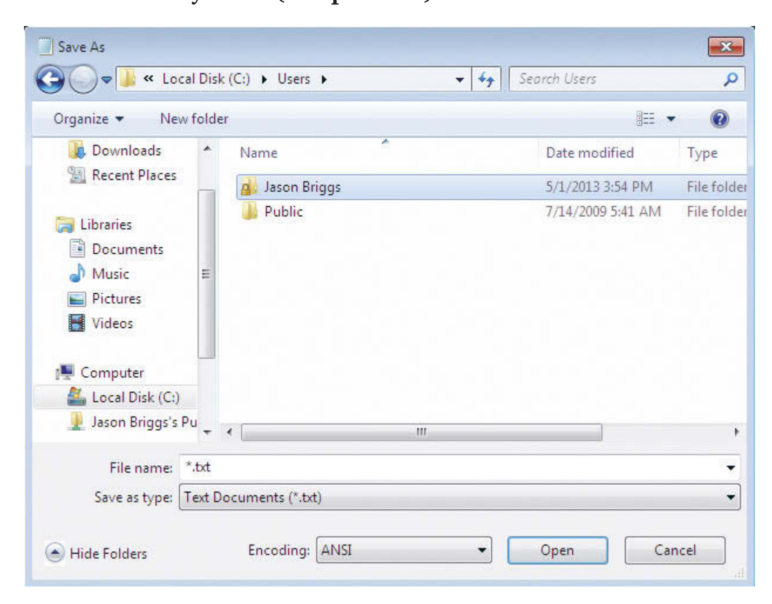
3. Выберите в меню **File ▸ Save** (**Файл ▸ Сохранить**).

4. В диалоговом окне выберите диск *C:* дважды кликнув **My Computer** (**Компьютер**), а затем дважды кликнув **Local Disk (C:) (Локальный диск (C:))**.

5. Дважды кликните по папке **Users** (**Пользователи**), а затем по вашему имени пользователя.

6. В поле ввода **File name** (**Имя файла**), которое находится внизу окна, введите *test.txt*.

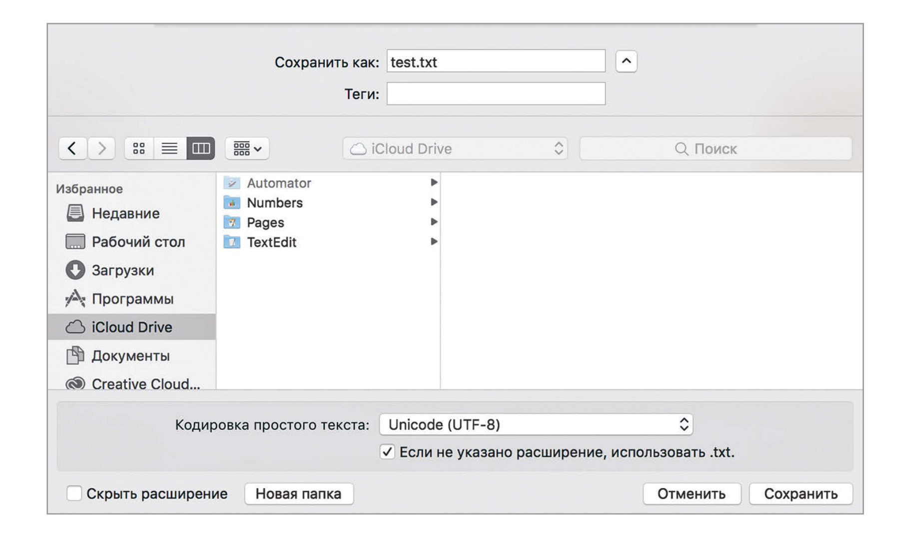
Нажмите кнопку **Save** (**Сохранить**).



Создание нового файла в Mac OS X

Выполните следующие шаги для создания файла *test.txt*:

1. Кликните по значку **Spotlight** в меню вверху экрана.
2. В появившейся строке поиска введите *TextEdit*.
3. В секции Applications должно появиться приложение TextEdit, кликните по нему (также TextEdit можно найти в Finder, в папке Applications).
4. Введите несколько строк текста.
5. Выберите **Format ▸ Make Plain Text**.
6. Выберите **File ▸ Save**.
7. В строке ввода **Save As** введите *test.txt*.
8. В списке Places кликните по вашему имени пользователя — имени, под которым вы вошли в систему, или имени человека, которому принадлежит этот компьютер.
9. Нажмите кнопку **Save**.



Создание нового файла в Ubuntu

Выполните следующие шаги для создания файла *test.txt*:

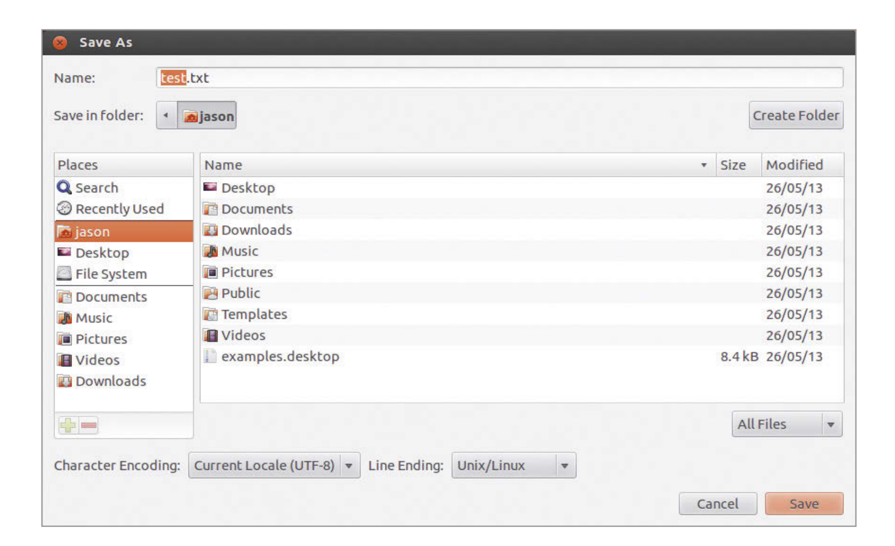
1. Откройте текстовый редактор, который обычно называется Text Editor. Если вы не пользовались им прежде, поищите его в меню **Applications**.

2. Введите несколько строк текста.

3. Выберите **File ▸ Save**.

4. В строке ввода **Name** введите *test.txt*. Скорее всего, ваша домашняя директория уже выбрана в строке **Save in Folder**. Если нет, найдите ее в списке Places и кликните по ней. (Название вашей домашней директории соответствует имени пользователя, под которым вы вошли в систему.)

5. Кликните по кнопке **Save**.



Открытие файла в Python

С помощью встроенной функции `open` можно открыть файл в оболочке Python и вывести его содержимое на экран. Этой функции нужно сообщить, где находится файл. Как это сделать, зависит от вашей операционной системы. Ознакомьтесь с примером открытия файла в Windows, а затем прочитайте параграфы, посвященные Mac и Ubuntu, если вы используете одну из этих систем.

Открытие файла в Windows

Для открытия файла *test.txt* введите:

Test file — пробный файл

```
>>> test_file = open('c:\\Users\\<ваше имя пользователя>\\
                     test.txt')
>>> text = test_file.read()
>>> print(text)
<здесь будет текст, который вы напечатали в блокноте>
```

В первой строке кода мы вызываем функцию `open`, которая возвращает файловый объект, содержащий функции для работы с файлом. В качестве аргумента для `open` мы передаем строку со сведениями о местонахождении файла (так называемый *путь к файлу*). Поскольку ваша система — Windows и вы сохранили файл в своей пользовательской папке на диске *C:*, путь выглядит так: `c:\\Users\\<ваше имя пользователя>\\test.txt`.

Двойные знаки «обратный слеш» означают, что это просто «слеш», а не специальная команда. (В главе 3 мы обсуждали, что в строках обратный слеш имеет особое значение.) Полученный файловый объект мы сохраняем в переменной `test_file`.

Далее мы вызываем функцию файлового объекта `read`, чтобы получить содержимое файла и сохранить это содержимое в переменной `text`. И наконец выводим полученные данные на экран.

Открытие файла в Mac OS X

Путь к файлу *test.txt* в первой строке кода также должен содержать имя пользователя, которое вы выбрали при записи файла. Например, если имя пользователя — *sarahwinters*, функцию `open` нужно вызывать так:

```
>>> test_file = open('/Users/sarahwinters/test.txt')
```

Открытие файла в Ubuntu

Если вы используете Ubuntu, путь к файлу *test.txt* в первой строке кода, как и в предыдущих двух случаях, должен содержать имя пользователя. Например, если имя пользователя — *jacob*, функцию `open` нужно вызывать так:

```
>>> test_file = open('/home/jacob/test.txt')
```

Запись в файл

Файловый объект, который возвращает функция open, предоставляет и другие функции помимо read (читать). Начнем с того, что при вызове open можно создать новый, пустой файл. Для этого вторым аргументом следует передать строку 'w':

```
>>> test_file = open('c:\\myfile.txt', 'w')
```

My file — мой файл

Аргумент 'w' означает, что мы собираемся записывать данные в наш файл, а не читать их.

Теперь можно записать в файл что-нибудь, воспользовавшись функцией write:

```
>>> test_file = open('c:\\myfile.txt', 'w')
>>> test_file.write('это - тестовый файл')
20
```

Write — *здесь* «записать (в файл)»

И наконец, нужно сообщить Python, что мы закончили писать в файл, — вызвать функцию close:

Close — закрыть

```
>>> test_file = open('c:\\myfile.txt', 'w')
>>> test_file.write('Что это - зеленое и крякает? Жабокряк!')
>>> test_file.close()
```

Если теперь открыть файл в текстовом редакторе, вы должны увидеть текст «Что это — зеленое и крякает? Жабокряк!». Также можно прочитать файл в оболочке Python:

```
>>> test_file = open('c:\\myfile.txt')
>>> print(test_file.read())
Что это - зеленое и крякает? Жабокряк!
```



Что мы узнали

В этой главе мы изучили встроенные функции языка Python: float и int, которые преобразуют вещественные числа в целые и обратно, len, которая полезна при создании циклов, а также функции для работы с файлами и некоторые другие.

Упражнения

Поэкспериментируйте с некоторыми из встроенных функций Python, выполнив эти упражнения.

#1. Таинственный код

Что выведет на экран следующий код? Попробуйте угадать, а затем запустите код и проверьте вашу догадку.

```
>>> a = abs(10) + abs(-10)
>>> print(a)
>>> b = abs(-10) + -10
>>> print(b)
```

#2. Зашифрованное сообщение

Help — помощь, поддержка

Постарайтесь с помощью функций dir и help узнать, как можно разбить строку на отдельные слова, а затем напишите небольшую программу, которая печатает слова из следующей строки через одно, начиная с первого слова («этот»):

```
"этот если способ вы плохо это подходит читаете для что-то ↵
шифрования пошло важных не сообщений так"
```

#3. Копирование файла

Напишите программу для копирования файла. (Подсказка: нужно открыть файл, который вы собираетесь скопировать, считать из него данные, создать новый файл-копию и записать туда считанные данные.) Проверьте результат работы программы, напечатав содержимое файла-копии на экране.

10

ПОЛЕЗНЫЕ МОДУЛИ PYTHON

Как вы уже знаете из главы 7, Python-модуль — это комплект функций, классов и переменных, собранных вместе для удобства использования. Например, модуль turtle, с которым мы имели дело в предыдущих главах, объединяет функции и классы для создания холста и рисования с помощью черепашки.

После того как модуль будет импортирован в программу, можно использовать все содержащиеся в нем элементы. Например, после импортирования модуля turtle в главе 4 мы получили доступ к классу Pen (ручка) и создали объект этого класса для рисования на экране:

```
>>> import turtle
>>> t = turtle.Pen()
```

В комплекте с Python идет множество модулей для выполнения самых разных задач. В этой главе мы рассмотрим и опробуем некоторые из наиболее полезных модулей.



Создание копий с помощью модуля copy

При написании программ обычно создаются новые объекты, но, если этот процесс проходит в несколько этапов, бывает удобно сделать копию уже существующего объекта. Для этого и нужен модуль copy (копировать).

Species — виды
Number of legs — количество ног
Color — цвет

Допустим, у нас есть класс Animal (животное) с функцией __init__, которая принимает аргументы species, number_of_legs и color.

```
>>> class Animal:
        def __init__(self, species, number_of_legs, color):
            self.species = species
            self.number_of_legs = number_of_legs
            self.color = color
```

Теперь можно создать новый объект класса Animal. Пусть это будет розовый гиппогриф с шестью ногами по имени Гарри (harry).

```
>>> harry = Animal('гиппогриф', 6, 'розовый')
```

Copy — копировать

Если нам нужна стайка шестиногих гиппогрифов, можно несколько раз повторить предыдущий код, а можно использовать функцию copy из модуля copy:

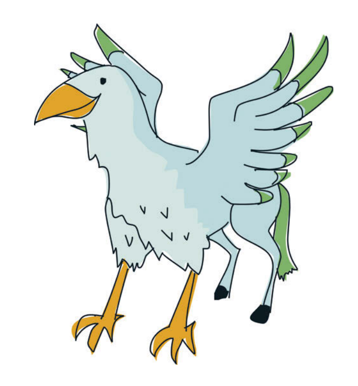
```
>>> import copy
>>> harry = Animal('гиппогриф', 6, 'розовый')
>>> harriet = copy.copy(harry)
>>> print(harry.species)
гиппогриф
>>> print(harriet.species)
гиппогриф
```

Мы создали объект, сохранив его в переменной harry, а затем — копию этого объекта, сохранив ее в переменной harriet (Гарриет). Данные объекты не зависят друг от друга, хотя и обладают одинаковыми свойствами.

Таким же образом можно создать и скопировать список объектов класса Animal.

Carrie — Кэрри
Billy — Билли
My animals — мои животные
More animals — больше животных

```
>>> harry = Animal('гиппогриф', 6, 'розовый')
>>> carrie = Animal('химера', 4, 'в зеленый горошек')
>>> billy = Animal('богл', 0, 'узорчатый')
>>> my_animals = [harry, carrie, billy]
>>> more_animals = copy.copy(my_animals)
>>> print(more_animals[0].species)
гиппогриф
>>> print(more_animals[1].species)
химера
```



В первых трех строках кода мы создали три объекта класса Animal, сохранив их в переменных harry, carrie и billy. В четвертой строке мы добавили их в список my_animals. Затем с помощью copy создали новый список more_animals и напечатали значения свойств species первых двух объектов ([0] и [1]) из этого списка. Обратите внимание: эти значения такие же, как у объектов из первоначального списка. То есть мы создали копию списка, не пересоздавая хранящиеся в нем объекты.

Однако, если заменить свойство species объекта «гиппогриф» из первоначального списка на «вампир», Python поменяет это свойство и у объекта из списка more_animals!

```
>>> my_animals[0].species = 'вампир'
>>> print(my_animals[0].species)
вампир
>>> print(more_animals[0].species)
вампир
```

Мы обращались к списку my_animals. Почему же изменились объекты в обоих списках?

Дело в том, что функция copy создает *поверхностную копию* — когда в копируемом объекте содержатся другие объекты, их копий она не делает. В данном случае copy создала копию списка, но не его элементов: список more_animals содержит те же три объекта, что и my_animals.

В то же время, если мы добавим новое животное в первый список (my_animals), оно не появится в списке-копии (more_animals). Чтобы в этом убедиться, после добавления животного выведем длину каждого из списков на экран:

Sally — Салли

```
>>> sally = Animal('сфинкс', 4, 'песочный')
>>> my_animals.append(sally)
>>> print(len(my_animals))
4
>>> print(len(more_animals))
3
```

Как видите, добавление объекта в список my_animals не повлияло на его копию — список more_animals. Функция len показала, что в первом списке четыре элемента, тогда как во втором — всего три.

В модуле copy есть другая функция — deepcopy, которая создает копии и для всех объектов внутри копируемого. Если скопировать список

Deep copy — глубокая копия

my_animals с помощью этой функции, мы получим новый список с копиями всех объектов, и тогда изменение элементов первоначального списка не повлияет на новый список. Например:

```
>>> more_animals = copy.deepcopy(my_animals)
>>> my_animals[0].species = 'дракон'
>>> print(my_animals[0].species)
дракон
>>> print(more_animals[0].species)
вампир
```

Когда мы поменяли свойство species первого объекта из первоначального списка, список-копия остался прежним. В этом можно убедиться, напечатав свойство species для первого объекта в каждом из списков.

Ключевые слова и модуль keyword

Keyword — ключевое слово

Is keyword — является ключевым словом

Kwlist — от keyword list — список ключевых слов

В Python *ключевым словом* называется слово, являющееся частью языка (например, if, else или for). В модуле keyword есть функция iskeyword (которая принимает строку и возвращает True, если в ней находится ключевое слово), а также переменная kwlist, в которой хранится список всех ключевых слов Python.

В следующем примере функция iskeyword возвращает True для строки if и False — для строки ozwald. При печати переменной kwlist мы видим полный список ключевых слов. Это может быть важно, поскольку в разных версиях Python ключевые слова могут различаться.

```
>>> import keyword
>>> print(keyword.iskeyword('if'))
True
>>> print(keyword.iskeyword('ozwald'))
False
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
'with', 'yield']
```

Описание всех ключевых слов можно найти в разделе «Приложение» на стр. 283.

Генерация случайных чисел с помощью модуля random

В модуле `random` содержится набор функций для генерации случайных чисел. Это примерно как попросить компьютер загадать число. Среди наиболее полезных — функции `randint`, `choice` и `shuffle`.

Random — случайный

Choice — выбор

Shuffle — перетасовка

Использование randint

Функция `randint` возвращает случайное число из заданного диапазона значений, скажем, от 1 до 100 или от 1000 до 5000. Например:

```
>>> import random
>>> print(random.randint(1, 100))
58
>>> print(random.randint(100, 1000))
861
>>> print(random.randint(1000, 5000))
3795
```

С помощью `randint` и цикла `while` можно написать простую игру «Угадай число»:

```
  >>> import random
  >>> num = random.randint(1, 100)
❶ >>> while True:
❷         print('Угадайте число от 1 до 100')
❸         guess = input()
❹         i = int(guess)
❺         if i == num:
              print('Правильно!')
❻             break
❼         elif i < num:
              print('Загаданное число больше')
❽         elif i > num:
              print('Загаданное число меньше')
```

Мы импортировали модуль `random` и сохранили в переменной `num` случайное число в диапазоне от 1 до 100, которое вернула функция `randint`. Затем в строке с меткой ❶ запустили бесконечный цикл `while` (точнее, он будет выполняться до тех пор, пока игрок не угадает число).

Num — от number — число

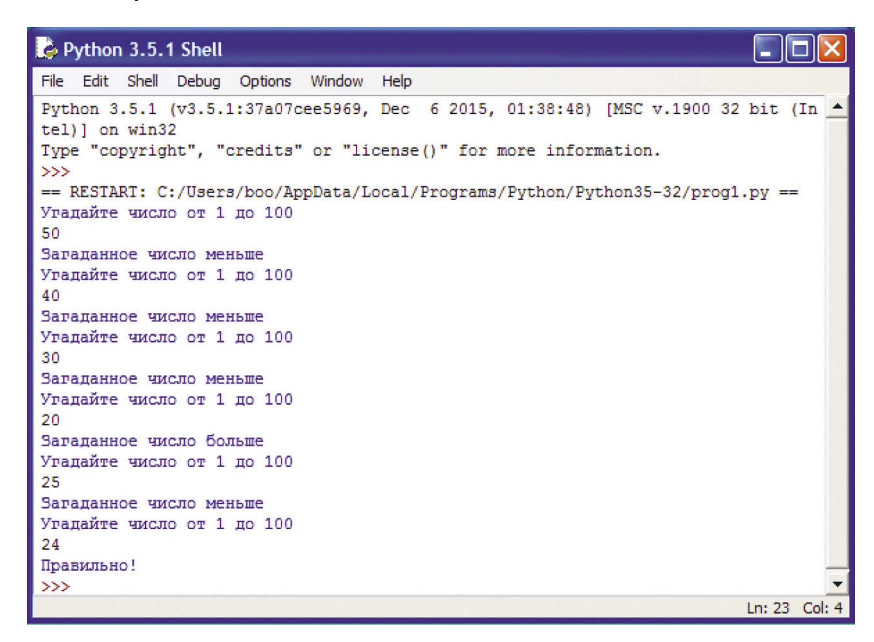
В строке ❷ печатаем сообщение «Угадайте число», в строке ❸ с помощью `input` считываем ответ пользователя, сохраняя его в переменной `guess` (догадка). В строке ❹ с помощью `int` преобразуем введенную строку в целое число, а в строке ❺ сравниваем его со случайным значением `num`.

Если введенное и случайно сгенерированное значения равны, печатаем «Правильно!» и выходим из цикла в строке ❻. В противном случае в строке ❼ проверяем, не меньше ли введенное число, чем загаданное, а в строке ❽, — не больше ли оно, выводя соответствующие сообщения.

Этот код не такой уж короткий, и у вас может возникнуть желание ввести его в новом окне оболочки либо создать новый текстовый файл с кодом, сохранить его и запустить в IDLE. Давайте вспомним, как открыть и запустить сохраненную программу:

1. Запустите IDLE и выберите **File ▸ Open**.
2. Найдите папку, где сохранен файл с программой, и кликните по имени файла.
3. Кликните **Open**.
4. Когда откроется новое окно, выберите в меню **Run ▸ Run Module**.

Запустив код последнего примера, вы должны увидеть приблизительно следующее:



Получение случайного элемента списка с помощью choice

Если требуется не сгенерировать случайное число из заданного диапазона, а выбрать случайный элемент списка, можно воспользоваться функцией `choice`. Python может выбрать для вас десерт, например кекс:

```
>>> import random
>>> desserts = ['мороженое', 'блинчики', 'кекс', 'печенье',
                'конфеты']
>>> print(random.choice(desserts))
кекс
```

Desserts — десерты

Перетасовка списка с помощью shuffle

Функция shuffle перетасовывает (случайным образом перемешивает) элементы списка. Если параллельно с чтением книги вы запускаете примеры в IDLE и уже импортировали модуль random, а также создали список desserts из предыдущего примера, можете сразу переходить к команде random.shuffle:

```
>>> import random
>>> desserts = ['мороженое', 'блинчики', 'пирог', 'печенье',
                'конфеты']
>>> random.shuffle(desserts)
>>> print(desserts)
['блинчики', 'печенье', 'пирог', 'конфеты', 'мороженое']
```

После вызова shuffle порядок элементов в списке не такой, как вначале. Функция shuffle может пригодиться, например, при создании карточной игры — для перетасовки списка, соответствующего карточной колоде.

Управление оболочкой с помощью модуля sys

Модуль sys содержит средства для взаимодействия с самим интерпретатором Python. Мы поговорим об использовании функции exit, объектов stdin и stdout, а также переменной version.

Выход из оболочки с помощью функции exit

Функция exit позволяет закрыть оболочку или консоль Python. Введите следующий код, и перед вами возникнет диалоговое окно с вопросом, действительно ли вы хотите выйти из оболочки. Кликните **Yes**, и оболочка закроется.

Exit — выход

Yes — да

```
>>> import sys
>>> sys.exit()
```

Однако это сработает только для модифицированной версии IDLE, про установку которой говорилось в главе 1, иначе вы увидите сообщение об ошибке:

```
>>> import sys
>>> sys.exit()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    sys.exit()
SystemExit
```

Ввод данных и объект stdin

Stdin — от standard input — стандартный ввод

Stdout — от standard output — стандартный вывод

Объект `stdin` из модуля `sys` служит для ввода информации в программу. Как нам известно из главы 7, у этого объекта есть функция `readline`, позволяющая ввести строку текста с клавиатуры — примерно как функция `input`, которую мы использовали в игре «Угадай число». Например, введите:

```
>>> import sys
>>> v = sys.stdin.readline()
Тот, кто смеется последним, медленнее соображает
```

Python сохранит строку «Тот, кто смеется последним, медленнее соображает» в переменной `v`. Чтобы в этом убедиться, напечатаем содержимое `v`:

```
>>> print(v)
Тот, кто смеется последним, медленнее соображает
```

Одно из различий между `input` и `readline` состоит в том, что для `readline` можно задать ограничение количества символов. Например:

```
>>> v = sys.stdin.readline(13)
Тот, кто смеется последним, медленнее соображает
>>> print(v)
Тот, кто смее
```

Вывод данных и объект stdout

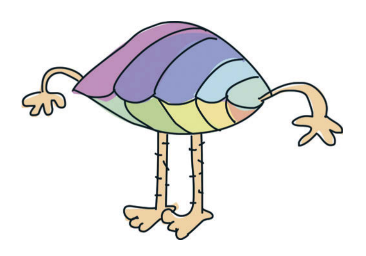
В отличие от stdin объект stdout предназначен для вывода сообщений в оболочке или консоли. Он позволяет делать то же, что и функция print, однако stdout — файловый объект, и у него есть файловые функции, о которых мы говорили в главе 9. Например, write:

```
>>> import sys
>>> sys.stdout.write("У какого слона нет хобота? У шахматного!")
У какого слона нет хобота? У шахматного!40
```

Функция write возвращает количество напечатанных символов, поэтому после сообщения и стоит число 40. Можно при каждом вызове write приплюсовывать это значение к переменной, чтобы всегда знать, сколько символов мы уже вывели на экран.

Какую версию Python я использую?

Переменная version содержит информацию о версии Python. Ее значение может пригодиться, в частности, для проверки, что на компьютере установлена последняя версия языка. Некоторые программисты предпочитают выводить информацию о версии Python при старте программы или отображать ее в окне «О программе»:



```
>>> import sys
>>> print(sys.version)
3.1.2 (r312:79149, Mar 21 2013, 00:41:52) [MSC v.1500 32 bit
(Intel)]
```

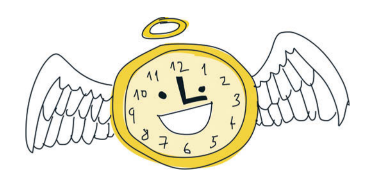
Version — версия

Работа со временем и модуль time

В модуле time есть функции для отображения времени, хотя порой результат их работы может вас удивить:

```
>>> import time
>>> print(time.time())
1300139149.34
```

Time — время



Функция time вернула количество секунд, прошедших от полуночи 1 января 1970 года до настоящего момента. Само по себе это число не кажется особо полезным, однако оно может пригодиться, например, для замера времени выполнения функции. Сохраним значение time до вызова функции и сразу после, а затем сравним эти числа. Давайте посмотрим, сколько времени займет печать всех чисел от 0 до 999.

Создадим функцию такого вида:

Lots of numbers — много чисел

```
>>> def lots_of_numbers(max):
        for x in range(0, max):
            print(x)
```

Проверим ее работу, передав 1000 в качестве аргумента max:

```
>>> lots_of_numbers(1000)
```

А теперь, чтобы узнать время работы функции, добавим в ее код вызовы time:

```
    >>> def lots_of_numbers(max):
❶           t1 = time.time()
❷           for x in range(0, max):
                print(x)
❸           t2 = time.time()
❹           print('Прошло %s секунд' % (t2-t1))
```

Запустив функцию еще раз, получим следующий результат (у вас он может быть другим, поскольку зависит от мощности компьютера):

```
>>> lots_of_numbers(1000)
0
1
2
3
.
.
.
997
998
999
Прошло 7.231045246124268 секунд
```

Вот как это работает: вызвав функцию time в первый раз, мы сохранили значение, которое она вернула, в переменной t1 (строка с меткой ❶). Затем в цикле, который начинается со строки ❷, напечатали числа от 0 до 999. После цикла, в строке ❸, снова вызвали time, сохранив результат в переменной t2. Поскольку работа цикла занимает определенное время, значение переменной t2 (количество секунд, прошедших с 1 января 1970 года) будет больше, чем значение t1. Вычтя t1 из t2 в строке ❹, мы узнали, сколько секунд потребовалось на печать всех чисел.

Преобразование дат с помощью asctime

Функция asctime принимает дату, заданную в виде кортежа, и преобразует ее в более понятный вид (напомню, что кортеж похож на список, но его элементы нельзя менять). Как мы уже видели в главе 7, вызов asctime без аргументов возвращает текущие дату и время в читаемом виде.

Asc time — узнать время

```
>>> import time
>>> print(time.asctime())
Mon Mar 11 22:03:41 2013
```

Mon Tue Wed Thu Fri Sat Sun — дни недели (первые три буквы), начиная с понедельника

Чтобы вызвать asctime с аргументом, создадим кортеж с параметрами даты и времени, сохранив его в переменной t:

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec — месяцы (первые три буквы), начиная с января

```
>>> t = (2020, 2, 23, 10, 30, 48, 6, 0, 0)
```

Значения в кортеже соответствуют году, месяцу, дню, часам, минутам, секундам, дню в неделе (0 — понедельник, 1 — вторник и так далее), дню в году (мы указали 0 в качестве «заглушки») и режиму летнего времени (0 — нет, 1 — да). Передав этот кортеж функции asctime, получим:

```
>>> import time
>>> t = (2020, 2, 23, 10, 30, 48, 6, 0, 0)
>>> print(time.asctime(t))
Sun Feb 23 10:30:48 2020
```

Получаем дату и время с помощью localtime

Функция localtime возвращает текущие дату и время в виде объекта, содержащего значения примерно того же вида, что и кортеж, который принимает функция asctime. Напечатав объект, мы увидим имя класса, к которому он принадлежит, а также перечень значений, помеченных

Local time — местное время

Year — год
Month — месяц
Day — день
Hour — час
Minute — минута
Second — секунда

как tm_year (год), tm_mon (месяц), tm_mday (день месяца), tm_hour (час) и так далее.

```
>>> import time
>>> print(time.localtime())
time.struct_time(tm_year=2020, tm_mon=2, tm_mday=23, tm_hour=22,
tm_min=18, tm_sec=39, tm_wday=0, tm_yday=73, tm_isdst=0)
```

Чтобы напечатать только текущие год и месяц, нужно получить соответствующие значения по их индексу (аналогично индексам в кортеже для asctime). Год идет первым (индекс = 0), а месяц вторым (индекс = 1). Следовательно, мы можем написать: year = t[0], month = t[1]:

```
>>> t = time.localtime()
>>> year = t[0]
>>> month = t[1]
>>> print(year)
2020
>>> print(month)
2
```

И видим, что сейчас второй месяц 2020 года.

Делаем передышку с помощью sleep

Sleep — сон

Функция sleep нужна, чтобы замедлить выполнение программы или приостановить ее. Например, для печати чисел от 1 до 60 можно использовать такой цикл:




```
>>> for x in range(1, 61):
        print(x)
```

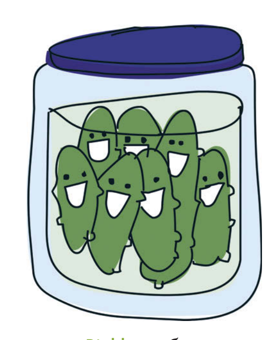
Этот код быстро напечатает все числа. Однако мы можем задать секундную паузу после вывода каждого числа. Вот так:

```
>>> for x in range(1, 61):
        print(x)
        time.sleep(1)
```

Числа будут выводиться раз в секунду. В главе 12 мы воспользуемся функцией sleep для создания реалистичной анимации.

Модуль pickle и сохранение информации

Модуль pickle используется для преобразования объектов Python в формат, удобный для записи в файл и последующего считывания из файла. Например, pickle пригодится, если вы разрабатываете игру и хотите записывать ее состояние на диск. Предположим, что у нас есть информация о состоянии игры в следующем виде:



Pickle — *букв.* засолить

Game data — данные игры

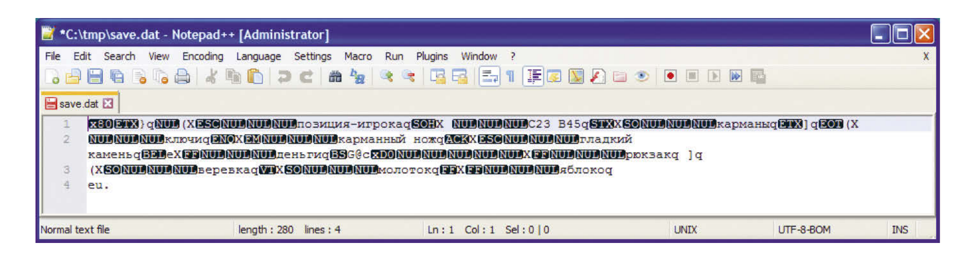
```
>>> game_data = {
        'позиция-игрока' : 'C23 B45',
        'карманы' : ['ключи', 'карманный нож', 'гладкий камень'],
        'рюкзак' : ['веревка', 'молоток', 'яблоко'],
        'деньги' : 158.50
}
```

Мы создали словарь, в котором содержится позиция игрока в нашей воображаемой игре, список предметов, которые находятся у игрока в кармане и рюкзаке, а также количество денег. Теперь можно сохранить этот словарь в файл, открыв этот файл для записи и воспользовавшись функцией dump из модуля pickle:

```
❶ >>> import pickle
❷ >>> game_data = {
        'позиция-игрока' : 'C23 B45',
        'карманы' : ['ключи', 'карманный нож', 'гладкий камень'],
        'рюкзак' : ['веревка', 'молоток', 'яблоко'],
        'деньги' : 158.50
        }
❸ >>> save_file = open('save.dat', 'wb')
❹ >>> pickle.dump(game_data, save_file)
❺ >>> save_file.close()
```

В строке ❶ мы импортировали модуль pickle, в строке ❷ создали словарь с данными игры. В строке ❸ открываем для записи файл *save.dat*, указав аргумент wb, который означает, что мы хотим записывать данные в бинарном режиме (возможно, файл потребуется создать в пользовательской папке, указав первым аргументом open путь вроде */Users/malcolmozwald/save.dat, /home/susanb/save.dat или C:\\Users\JimmyIpswich\save.dat*, как мы делали в главе 9). В строке ❹ вызываем функцию dump, передавая ей два аргумента: наш словарь и переменную с файловым объектом. В строке ❺ закрываем файл, поскольку больше с ним ничего делать не надо.

В текстовых файлах содержатся символы, понятные человеку. Однако изображения, аудио-, видеофайлы, а также объекты Python, сохраненные с помощью `pickle`, обычно содержат информацию, которую человек прочитать не может. Их называют бинарными (двоичными) файлами. Если вы откроете файл save.dat в текстовом редакторе, то увидите, что он не похож на текстовый файл, а представляет собой беспорядочную смесь текста и специальных символов.



Load — загрузить

Load file — загрузить файл

Loaded game data — загруженные данные игры

Rb — от read binary — прочитать в бинарном режиме

Объекты, записанные с помощью `pickle`, можно снова загрузить. Для этого понадобится `pickle`-функция `load`. Данные из файла будут преобразованы обратно в значения, которыми может оперировать наша программа. В целом `load` используется примерно так же, как `dump`:

```
>>> load_file = open('save.dat', 'rb')
>>> loaded_game_data = pickle.load(load_file)
>>> load_file.close()
```

Открываем файл с аргументом `rb`, что означает чтение файла в бинарном режиме. Вызываем функцию `load`, передавая ей файловый объект, а результат ее работы сохраняем в переменной `loaded_game_data` и закрываем файл.

Теперь можно убедиться, что данные загрузились без ошибок, напечатав значение переменной:

```
>>> print(loaded_game_data)
{'позиция-игрока': 'C23 B45', 'карманы': ['ключи', 'карманный
нож', 'гладкий камень'], 'деньги': 158.5, 'рюкзак': ['веревка',
'молоток', 'яблоко']}
```

Что мы узнали

В этой главе мы узнали, что в модулях Python содержатся функции, классы и переменные, и выяснили, что их можно использовать, импортируя модули в программу. Научились копировать объекты, генерировать случайные числа, перетасовывать элементы списка, работать со временем и датами, а также записывать и загружать данные с помощью `pickle`.

Упражнения

Попрактикуйтесь в использовании модулей Python, выполнив эти задания.

#1. Копирование

Что выведет на экран этот код?

```
>>> import copy
>>> class Car:
        pass

>>> car1 = Car()
>>> car1.wheels = 4
>>> car2 = car1
>>> car2.wheels = 3
>>> print(car1.wheels)
```

Что появится на экране?

```
>>> car3 = copy.copy(car1)
>>> car3.wheels = 6
>>> print(car1.wheels)
```

Что появится на экране?

Car — машина

Wheels — колеса

#2. Запись и загрузка

Создайте список ваших любимых вещей, а затем с помощью `pickle` запишите его в файл *favorites.dat*. Закройте оболочку Python, запустите ее снова и напечатайте содержимое вашего списка, загрузив его из файла.

Favorites — любимые



И СНОВА ЧЕРЕПАШЬЯ ГРАФИКА

Давайте вернемся к модулю turtle, с которым мы познакомились в главе 4. Как станет ясно из этой главы, в языке Python черепашки способны на кое-что большее, чем рисование простых черных линий. К примеру, с их помощью можно изображать сложные геометрические фигуры, смешивать цвета и заполнять цветом контуры фигур.

Начнем с обычного квадрата

Мы уже знаем, как рисовать линии с помощью черепашки. Нужно импортировать модуль turtle и создать объект Pen:

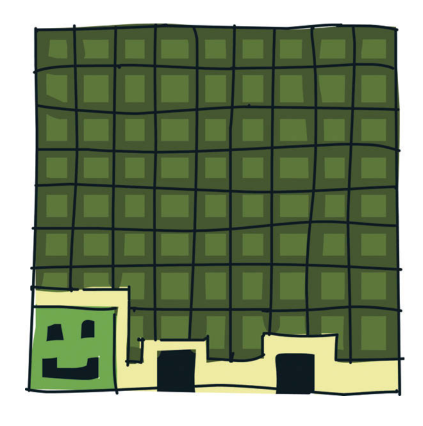
```
>>> import turtle
>>> t = turtle.Pen()
```

Вспомните код для рисования квадрата из главы 4:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
```

В главе 6 мы узнали о циклах, и теперь можем упростить этот состоящий из набора одинаковых команд код, воспользовавшись циклом `for`. Вот так:

```
>>> t.reset()
>>> for x in range(1, 5):
        t.forward(50)
        t.left(90)
```



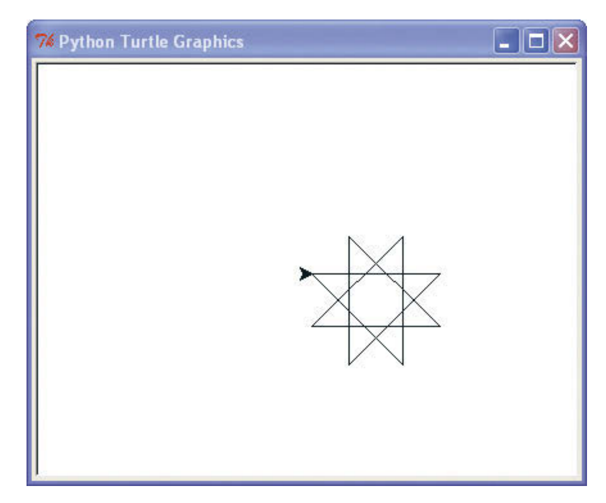
Устанавливаем объект `Pen` (черепашку) в начальную позицию командой `reset`. Задаем цикл от 1 до 4 с помощью `range(1, 5)`. На каждом повторе цикла перемещаем черепашку вперед на 50 пикселей и разворачиваем ее на 90 градусов влево. Благодаря циклу код стал заметно короче: если не считать вызов `reset`, он занимает три строки вместо шести.

Рисуем звезды

Можно изобразить кое-что поинтереснее, немного изменив цикл. Введите следующий код:

```
>>> t.reset()
>>> for x in range(1, 9):
        t.forward(100)
        t.left(225)
```

И получится восьмиконечная звезда:



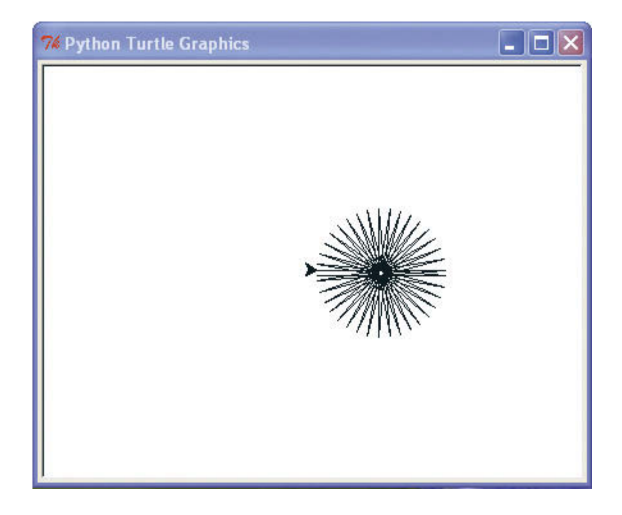
Этот код очень похож на код для квадрата, но есть отличия:

- вместо четырех повторов (range(1, 5)) делаем восемь (range(1, 9));
- перемещаем черепашку не на 50, а на 100 пикселей вперед;
- разворачиваем черепашку на 225 градусов, а не на 100.

Теперь давайте изобразим звезду с множеством лучей. Для этого достаточно указать угол в 175 градусов и сделать в цикле 37 повторов:

```
>>> t.reset()
>>> for x in range(1, 38):
        t.forward(100)
        t.left(175)
```

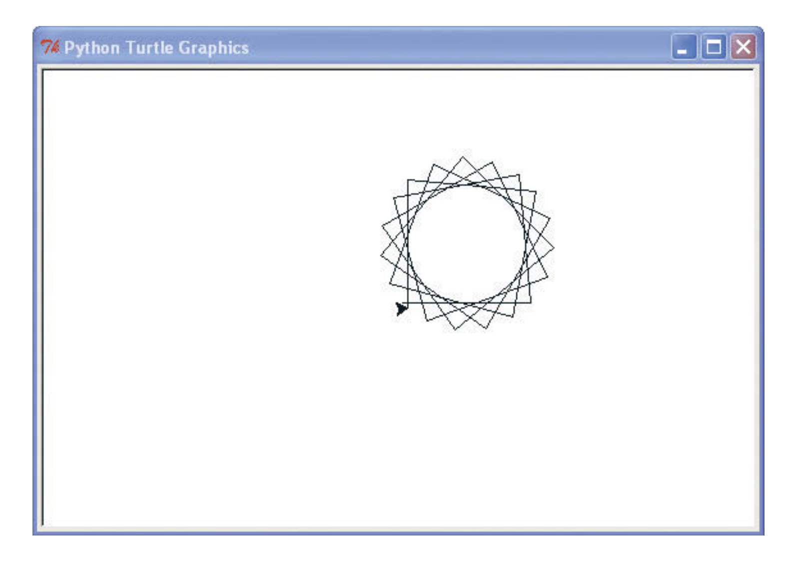
Вот результат работы этого кода:



И раз уж мы занялись звездами, вот код для рисования спиралевидной звезды:

```
>>> t.reset()
>>> for x in range(1, 20):
        t.forward(100)
        t.left(95)
```

Мы снова поменяли угол и сократили количество повторов, чтобы получилась звезда такого вида:

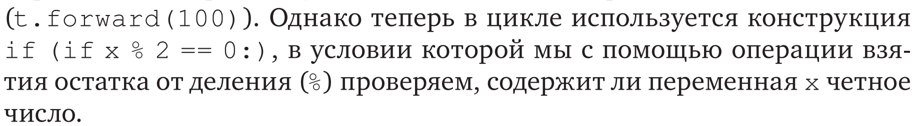


С помощью однотипного кода, где меняется лишь пара значений, можно нарисовать немало фигур — от квадрата до спиралевидной звезды. Благодаря циклу `for` это несложно, а без цикла пришлось бы много раз вводить одинаковые команды.

Теперь изобразим звезду другого типа, воспользовавшись конструкцией `if` для управления поворотами черепашки. В этом примере черепашка при четных повторах цикла разворачивается на 175 градусов, а при нечетных — на 225.

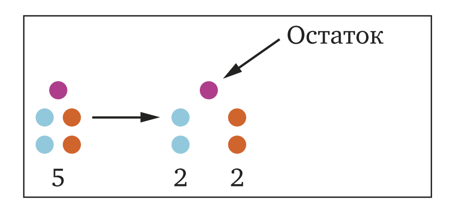


```
>>> t.reset()
>>> for x in range(1, 19):
        t.forward(100)
        if x % 2 == 0:
                t.left(175)
        else:
                t.left(225)
```



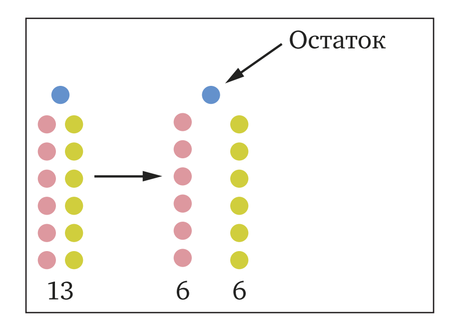
Мы задали цикл, тело которого повторяется 18 раз (`range(1, 19)`). При каждом повторе перемещаем черепашку на 100 пикселей вперед (`t.forward(100)`). Однако теперь в цикле используется конструкция `if` (`if x % 2 == 0:`), в условии которой мы с помощью операции взятия остатка от деления (`%`) проверяем, содержит ли переменная `x` четное число.

Выражение `x % 2` означает: что будет в остатке, если мы разделим число, хранящееся в переменной `x`, пополам? Например, если поделить



5 мячиков на две равные части, получится два раза по 2 мячика (всего 4) и еще один уйдет в остаток, как показано на рисунке.

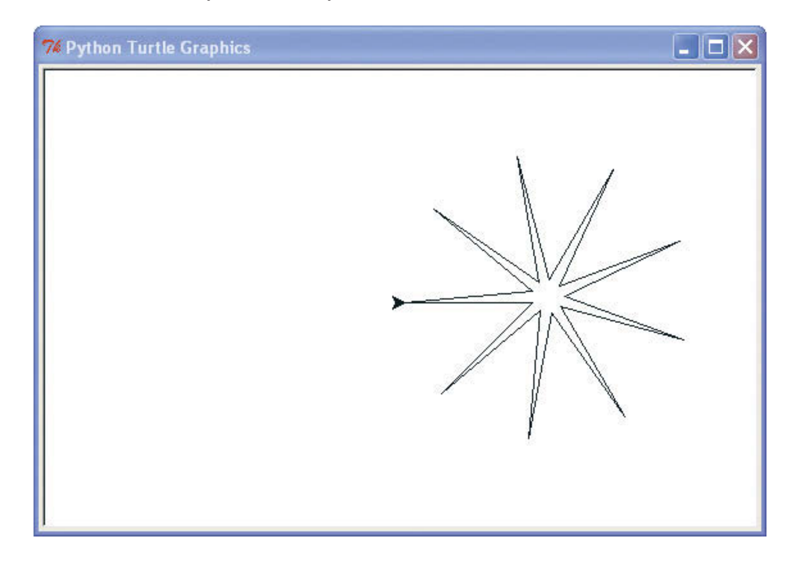
А если делить на две части 13 мячиков, получится 2 раза по 6 мячиков и один в остатке:



Вычисляя, равен ли нулю остаток от деления `x` на 2, мы, по сути, проверяем, можно ли поделить `x` на две части без остатка. Это удобный способ выяснить, является ли содержимое переменной четным числом, ведь четные числа всегда без остатка делятся на 2.

Итак, если значение `x` четное (`if x % 2 == 0:`), даем черепашке команду развернуться на 175 градусов влево (`t.left(175)`). Если нечетное (`else`), то на 225 градусов (`t.left(225)`).

И вот что мы получим, запустив наш код:



Рисуем машину

Черепашка может рисовать не только звезды и простые геометрические фигуры. Давайте изобразим простенькую, но симпатичную машинку. Сначала нарисуем кузов. Выберите в меню IDLE **File ▸ New File** и введите в новом окне следующий код:

```
import turtle
t = turtle.Pen()
t.reset()
t.color(1,0,0)
t.begin_fill()
t.forward(100)
t.left(90)
t.forward(20)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(60)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(20)
t.end_fill()
```

Теперь добавим переднее колесо:

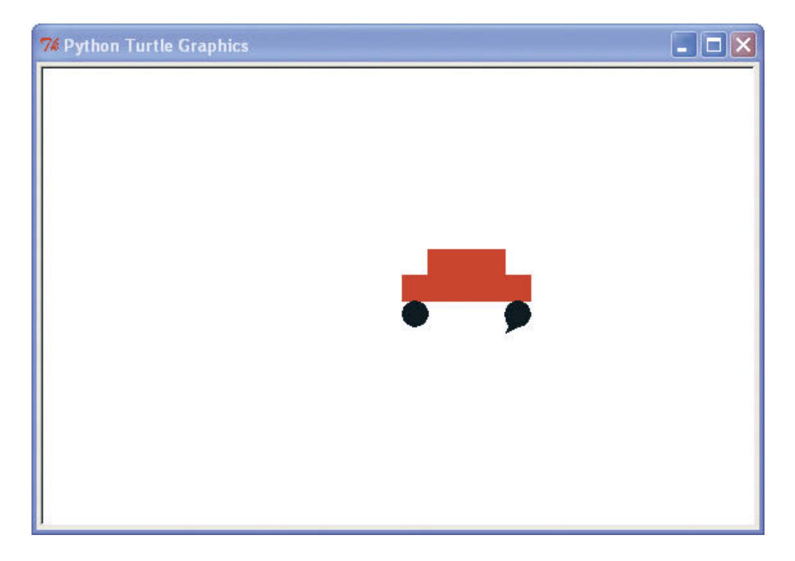
```
t.color(0,0,0)
t.up()
t.forward(10)
t.down()
t.begin_fill()
t.circle(10)
t.end_fill()
```

И заднее колесо:

```
t.setheading(0)
t.up()
t.forward(90)
t.right(90)
t.forward(10)
t.setheading(0)
```

```
t.begin_fill()
t.down()
t.circle(10)
t.end_fill()
```

Выберите **File ▸ Save As** и назовите файл *car.py*. Теперь выберите **Run ▸ Run Module**, чтобы запустить программу. И вот она, наша машинка:



Наверное, вы заметили, что в этом коде появилось несколько новых команд для черепашки:

Color — цвет

Begin fill — начать заполнение

End fill — закончить заполнение

Circle — круг

Set heading — здесь «установить направление»

- с помощью `color` можно задать цвет рисования;
- команды `begin_fill` и `end_fill` позволяют заполнить цветом область на холсте;
- `circle` рисует окружность указанного размера;
- `setheading` разворачивает черепашку так, чтобы она смотрела в указанном направлении.

Давайте разберемся, как с помощью этих функций сделать рисунки цветными.

Возьмемся за краски

Функция `color` принимает три аргумента. Первый определяет количество красного цвета, второй — зеленого, а третий синего. Например,

чтобы раскрасить кузов машины в ярко-красный, мы использовали команду `color(1,0,0)`, что означает 100-процентную насыщенность красного цвета.

Этот способ смешивания цветов на основе красного, зеленого и синего называется *цветовой моделью RGB*. Так отображает цвета монитор вашего компьютера: разные пропорции красного, зеленого и синего позволяют получить остальные цвета (примерно так же, как смешение синей и красной краски дает фиолетовую или как из желтой и красной выходит оранжевая). Красный, зеленый и синий цвета называют основными, поскольку из них получают остальные цвета.

RGB — от red, green, blue — красный, зеленый, синий

Хотя для отображения цвета на экране монитора используется не краска, а свет, представим, что RGB-модель состоит из трех банок с краской: красной, зеленой и синей. Каждая банка наполнена до краев. Будем считать, что полной банке соответствует число 1 (или 100 процентов). Теперь берем всю красную краску и всю зеленую, смешиваем и получаем желтый цвет (1 и 1, или по 100 процентов каждого цвета).

Вернемся к программному коду. Чтобы нарисовать желтую окружность, нам нужно по 100 процентов красного и зеленого цвета, а синий не нужен:

```
>>> t.color(1,1,0)
>>> t.begin_fill()
>>> t.circle(50)
>>> t.end_fill()
```

Числа 1, 1, 0 в первой строке соответствуют 100 процентам красного, 100 процентам зеленого и 0 процентам синего. В следующей строке мы говорим, что будем заполнять этим RGB-цветом контуры, которые нарисует черепашка (`t.begin_fill`), затем даем команду нарисовать окружность (`t.circle`). Команда `end_fill` в последней строке указывает, что окружность нужно заполнить выбранным RGB-цветом.

Функция для рисования заполненной окружности

Чтобы экспериментировать с цветом было проще, оформим код для рисования заполненной окружности в виде отдельной функции:

```
>>> def mycircle(red, green, blue):
        t.color(red, green, blue)
        t.begin_fill()
        t.circle(50)
        t.end_fill()
```

My circle — мой круг

Чтобы изобразить ярко-зеленый круг, достаточно дать такую команду:

```
>>> mycircle(0, 1, 0)
```

А для рисования темно-зеленого круга используем половину (0.5) зеленой краски. Вот так:

```
>>> mycircle(0, 0.5, 0)
```

Продолжим исследовать RGB-цвета. Изобразим круг, используя всю красную краску (1), затем половину (0.5), потом всю синюю, и наконец, половину синей:

```
>>> mycircle(1, 0, 0)
>>> mycircle(0.5, 0, 0)
>>> mycircle(0, 0, 1)
>>> mycircle(0, 0, 0.5)
```

! *С помощью команды* `t.reset()` *холст очищается от старых рисунков. И не забывайте, что черепашку можно перемещать без рисования линий: используйте* `t.up()`*, чтобы отключить рисование, и* `t.down()` *— чтобы включить.*

Различные комбинации красного, зеленого и синего цвета позволяют получить всевозможные оттенки, например золотой:

```
>>> mycircle(0.9, 0.75, 0)
```

Или светло-розовый:

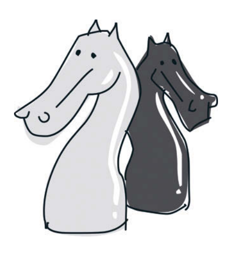
```
>>> mycircle(1, 0.7, 0.75)
```

А вот два оттенка оранжевого:

```
>>> mycircle(1, 0.5, 0)
>>> mycircle(0.9, 0.5, 0.15)
```

Попробуйте посмешивать цвета сами!

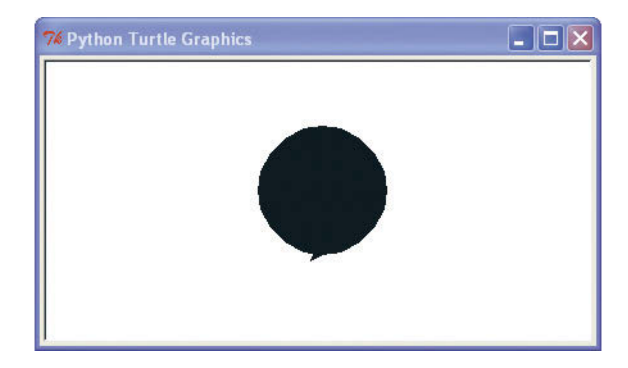
Получение черного и белого цвета



Что будет, если посреди ночи выключить свет? Все станет черным. Так же и на экране монитора: нет света — не будет и цвета. Поэтому, указав 0 для всех основных цветов, мы получим черный:

```
>>> mycircle(0, 0, 0)
```

Вот результат:



Если же смешать по 100 процентов всех основных цветов, результат будет противоположным — получится белый цвет. Введите следующий код, и белый круг сотрет с холста нарисованный ранее черный:

```
>>> mycircle(1, 1, 1)
```

Функция рисования квадрата

Для рисования заполненных цветом фигур нужно включить режим заполнения командой `begin_fill` и рисовать фигуры. Однако заполнение начнется только при вызове функции `end_fill`. Давайте еще немного поэкспериментируем с рисованием и заполнением. Возьмем функцию рисования квадрата из начала этой главы и сделаем так, чтобы она принимала размер стороны квадрата в качестве аргумента.

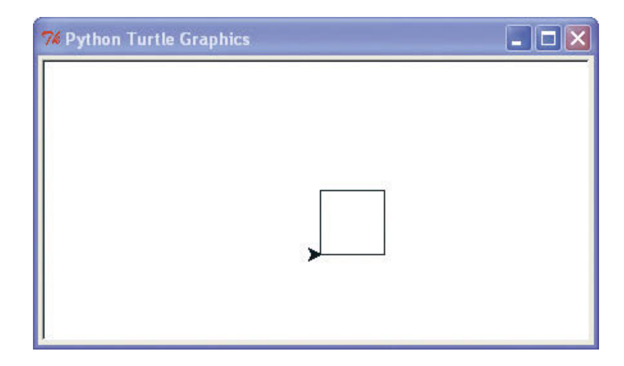
```
>>> def mysquare(size):
        for x in range(1, 5):
            t.forward(size)
            t.left(90)
```

My square — мой квадрат
Size — размер

Проверим работу функции, вызвав ее с аргументом 50:

```
>>> mysquare(50)
```

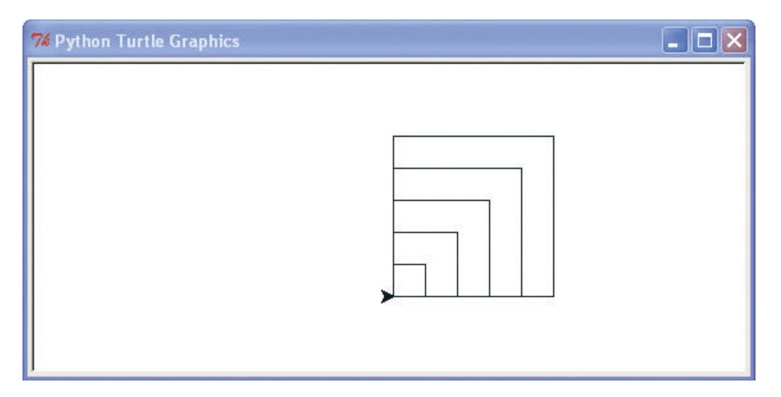
Получится квадратик:



Посмотрим, что выйдет, если передавать в нашу функцию разные значения. Следующий код изобразит пять вложенных квадратов со сторонами 25, 50, 75, 100 и 125.

```
>>> t.reset()
>>> mysquare(25)
>>> mysquare(50)
>>> mysquare(75)
>>> mysquare(100)
>>> mysquare(125)
```

Вот что должно получиться:



Рисуем заполненные квадраты

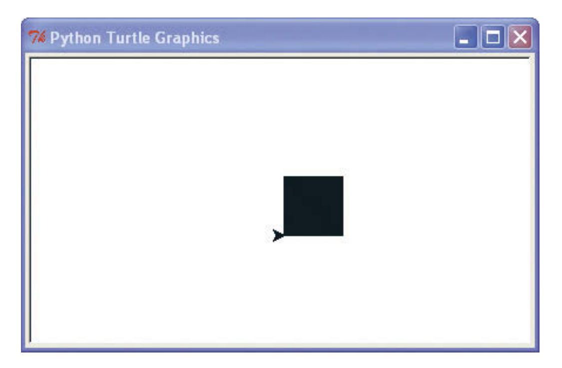
Изобразим заполненный квадрат. Очистим холст, включим режим заполнения и вызовем функцию еще раз:

```
>>> t.reset()
>>> t.begin_fill()
>>> mysquare(50)
```

Получившийся квадратик будет пустым, пока мы не вызовем функцию `end_fill`:

```
>>> t.end_fill()
```

В результате квадрат станет таким:



Изменим функцию так, чтобы одним ее вызовом можно было нарисовать либо заполненный квадрат, либо только контур. Для этого придется добавить еще один аргумент и проверять его значение в теле функции, так что ее код станет чуть сложнее:

```
>>> def mysquare(size, filled):
        if filled == True:
            t.begin_fill()
        for x in range(1, 5):
            t.forward(size)
            t.left(90)
        if filled == True:
            t.end_fill()
```

Filled — закрашен

В первой строке переопределяем функцию, чтобы она принимала два аргумента: `size` и `filled`. В коде функции будем проверять, равен ли

аргумент filled значению True (filled == True). Если равен, вызываем функцию begin_fill, которая включает режим заполнения. Затем в четырех повторах цикла (for x in range(0, 4)) рисуем четыре стороны квадрата (каждый раз перемещая черепашку вперед и разворачивая влево). Проверяем filled на равенство True, вызывая t.end_fill, если это так — в результате контур закрасится, а режим заполнения отключится.

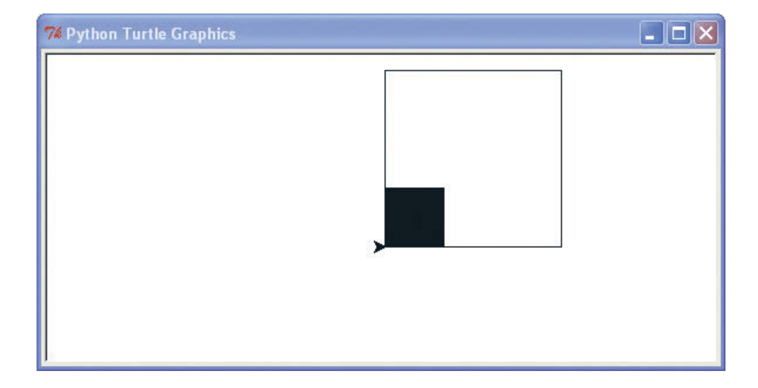
Теперь, чтобы нарисовать закрашенный квадрат, достаточно одной команды:

```
>>> mysquare(50, True)
```

Или так, чтобы изобразить пустой контур:

```
>>> mysquare(150, False)
```

В результате этих двух вызовов получится следующая картинка, похожая на квадратный глаз:



И это еще не всё. Рисовать и заполнять цветом можно разные фигуры.

Рисуем закрашенные звезды

В качестве завершающего примера снова изобразим звезду, но заполненную цветом. Наш прежний код выглядел так:

```
for x in range(1, 19):
        t.forward(100)
        if x % 2 == 0:
              t.left(175)
        else:
              t.left(225)
```

Создадим функцию mystar, принимающую размер звезды и признак ее заполнения, использовав в коде две конструкции if из функции mysquare:

My star — моя звезда

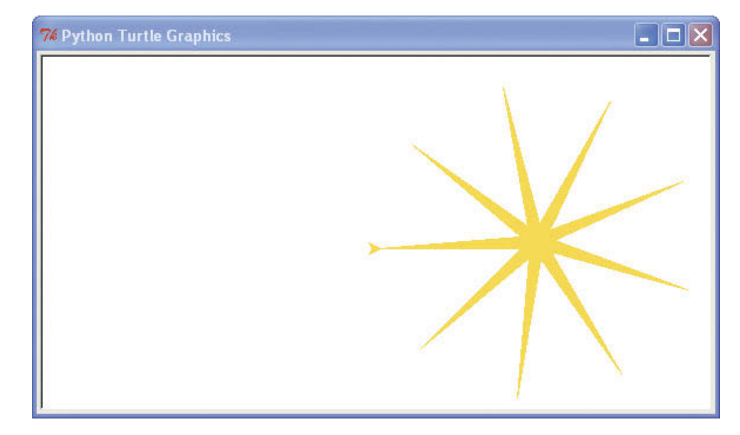
```
>>> def mystar(size, filled):
        if filled == True:
           t.begin_fill()
        for x in range(1, 19):
           t.forward(size)
           if x % 2 == 0:
              t.left(175)
           else:
              t.left(225)
        if filled == True:
              t.end_fill()
```

В первых двух строках кода проверяем filled на равенство True и включаем режим заполнения. Аналогичная проверка выполняется и в последних двух строках, где мы завершаем заполнение, если filled равно True. Так же, как и для функции mysquare, размер звезды передаем в аргументе size, используя это значение при вызове t.forward.

Включим золотой цвет рисования (90 процентов красного, 75 — зеленого, 0 — синего) и вызовем нашу функцию:

```
>>> t.color(0.9, 0.75, 0)
>>> mystar(120, True)
```

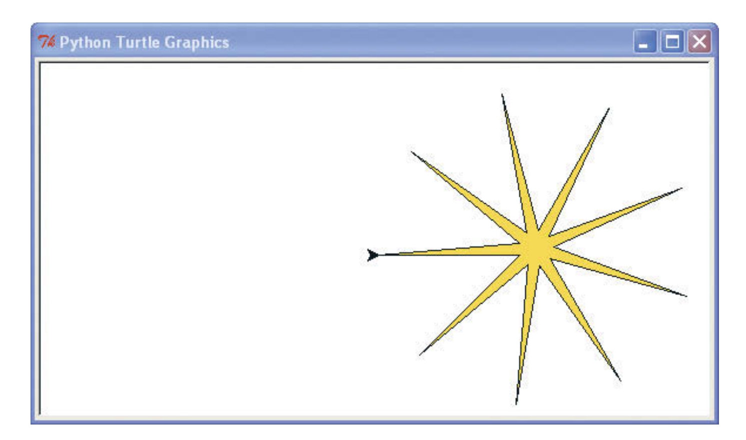
Черепашка нарисует заполненную звезду:



Обведем ее по контуру, включив черный цвет и вызвав функцию снова (на этот раз без заполнения):

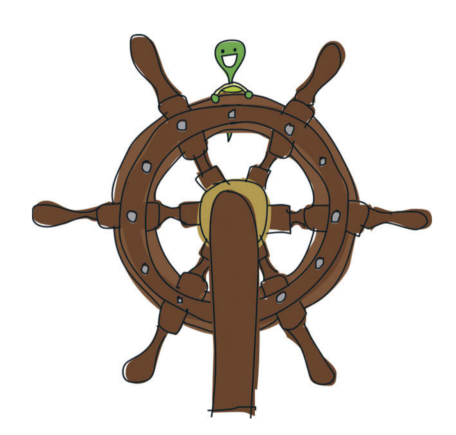
```
>>> t.color(0,0,0)
>>> mystar(120, False)
```

Получится золотая звезда с черной обводкой:



Что мы узнали

В этой главе мы узнали, как использовать модуль `turtle` для рисования некоторых простых геометрических фигур, управляя движениями черепашки с помощью циклов `for` и конструкций `if`. Научились менять цвет пера черепашки и заполнять этим цветом контуры. Также мы создали несколько функций, позволяющих изображать разноцветные фигуры одной командой.

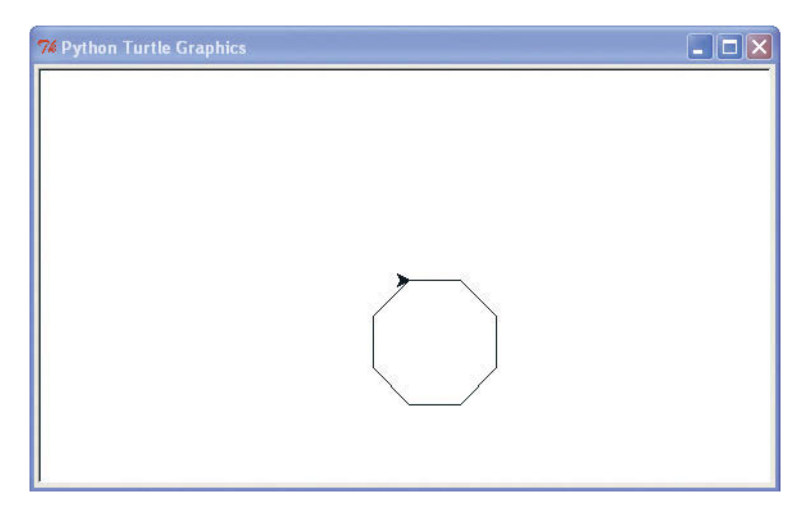


Упражнения

Выполните упражнения на рисование фигур с помощью черепашки.

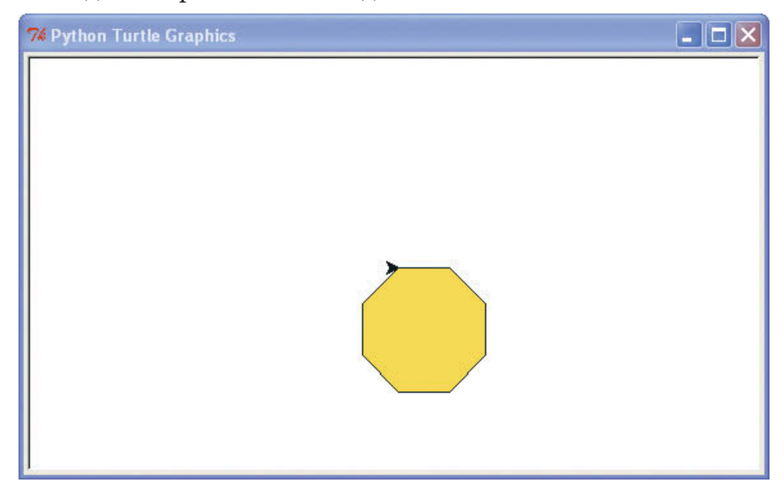
#1. Рисуем восьмиугольник

Мы уже умеем рисовать звезды, квадраты и прямоугольники. А теперь создадим функцию для рисования восьмиугольника. (Подсказка: для поворота черепашки используйте угол 45 градусов.)



#2. Заполненный восьмиугольник

Измените функцию для рисования восьмиугольника так, чтобы она изображала восьмиугольник заполненным. Попробуйте обвести его контур, как мы это делали раньше со звездой.



#3. Еще одна функция для рисования звезд

Создайте функцию для рисования звезд, которая принимает два аргумента: размер и количество точек, между которыми проведены линии, составляющие фигуру. Определение этой функции будет начинаться примерно так:

Draw star — нарисовать звезду

Points — точки

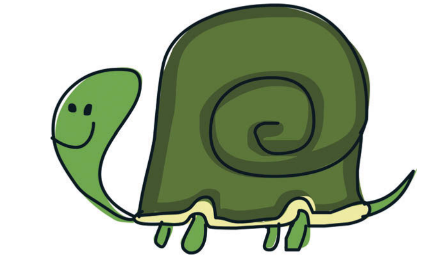
```
def draw_star(size, points):
```

12

БОЛЕЕ СОВЕРШЕННАЯ ГРАФИКА С МОДУЛЕМ TKINTER

Рисование с помощью черепашки имеет один недостаток: черепашки… передвигаются... очень… медленно. Даже рекордная черепашья скорость оставляет желать лучшего. Черепахам это особых неудобств не доставляет, зато сильно ограничивает возможности компьютерной графики.

Компьютерная графика, особенно в играх, требует быстрой отрисовки. Если у вас есть игровая приставка или вы играете в видеоигры на компьютере, подумайте о том изображении, которое видите на экране. В двухмерных (2D) играх изображение плоское, персонажи двигаются в основном вверх, вниз, вправо и влево — как в большинстве игр для Nintendo DS, PlayStation Portable (PSP) и мобильных телефонов. В псевдотрехмерных играх, которые выглядят почти объемными, изображение более реалистичное, но персонажи обычно перемещаются в одной плоскости (это называется *изометрической графикой*). И наконец, есть трехмерные игры, графика в которых стремится к максимальной реалистичности. Однако все игры — двухмерные, псевдотрехмерные и трехмерные — имеют общую черту: изображения на экран выводятся очень быстро.



Если вы никогда не занимались анимацией, попробуйте сделать вот что:

1. Возьмите чистый блокнот и нарисуйте что-нибудь в нижнем углу первой страницы, (например, человечка).

2. В углу следующей страницы нарисуйте такого же человечка почти в той же позе, но слегка сдвиньте одну его ногу.

3. На следующей странице снова нарисуйте человечка, сдвинув его ногу еще немного.

4. Страница за страницей рисуйте в нижнем углу человечка, каждый раз чуть-чуть изменяя его позу.

Когда закончите, быстро пролистайте страницы, глядя на человечка, и увидите, как он движется. Этот способ лежит в основе любой анимации — телевизионной или компьютерной. На экран выводится картинка, потом выводится снова, но с небольшими изменениями, и так далее. Возникает иллюзия движения, при этом нужно, чтобы картинки (кадры анимации) сменяли друг друга очень быстро.

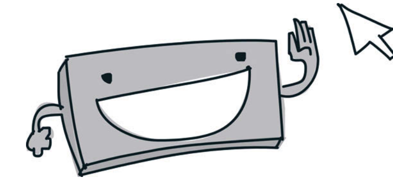
Есть разные способы работы с графикой в языке Python. Помимо модуля `turtle` можно использовать внешние модули (которые устанавливаются отдельно) или модуль `tkinter`, который по умолчанию ставится вместе с Python. Этот модуль позволяет создавать полноценные приложения, вроде несложного текстового редактора, а также работать с графикой. В данной главе мы узнаем, как создавать изображения и анимации средствами `tkinter`.

Создаем кнопку

В качестве первого примера воспользуемся `tkinter` для создания простейшего приложения с одной кнопкой. Введите следующий код:

From — из

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text="нажми меня")
>>> btn.pack()
```



В первой строке мы импортировали содержимое модуля `tkinter`. Конструкция `from <имя модуля> import *` позволяет обращаться к содержимому модуля, не указывая каждый раз его имя. Вспомните, как, используя `import turtle` в предыдущих примерах, мы обращались к функциям модуля через имя `turtle`:

```
import turtle
t = turtle.Pen()
```

Если воспользоваться `import *`, то не нужно писать `turtle.Pen`, как мы делали в главах 4 и 11. В случае с `turtle` особых преимуществ это не дает, но данный способ удобен при использовании модулей,

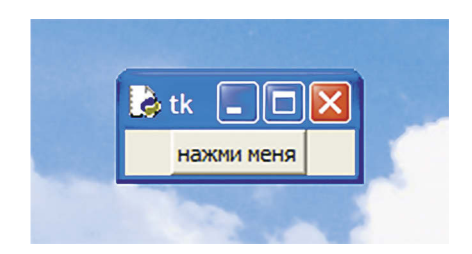
содержащих множество классов и функций, так как позволяет вводить меньше кода.

```
from turtle import *
t = Pen()
```

Во второй строке примера с кнопкой (`tk = Tk()`) мы создали переменную и сохранили в ней объект класса `Tk` (аналогично созданию объекта `Pen` для черепашки). Объект `tk` создает пустое окно, в которое можно добавлять кнопки, строки ввода, холсты для рисования и так далее. Это основной класс модуля `tkinter`. Без создания объекта класса `Tk` работать с графикой или анимацией с помощью этого модуля невозможно.

В третьей строке мы создали кнопку вызовом `btn = Button`, передав первым аргументом переменную `tk`, а вторым — строку `"нажми меня"` (текст, который нужно отобразить на кнопке). Несмотря на то что мы добавили кнопку в окно, она не появится на экране до ввода команды `btn.pack()`, которая включает ее отображение. Также эта команда управляет выравниванием кнопок и других графических элементов в окне. В результате получится примерно следующее:

Button — кнопка



От кнопки *нажми меня* не слишком много пользы — чтобы при нажатии на нее что-то происходило, код нужно изменить. (Перед этим не забудьте закрыть созданное ранее окно!)

Создадим функцию, которая выводит сообщение:

```
>>> def hello():
        print('привет')
```

Hello — привет

Изменим код примера, пользуясь этой функцией:

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text="нажми меня", command=hello)
>>> btn.pack()
```

Command — команда

Обратите внимание: мы лишь слегка изменили предыдущий код, добавив аргумент `command`, указывающий, что при клике мышкой по кнопке нужно вызывать функцию `hello`.

Если теперь кликнуть по кнопке, в окне оболочки появится сообщение «привет». Оно будет выводиться при каждом клике. На картинке ниже видно, что я нажал кнопку пять раз.

```
*Python 3.5.1 Shell*
File  Edit  Debug  Options  Window  Help
and will be removed in a later version. See Help/IDLE Help
for details.

>>> def hello():
        print('привет')


>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text="нажми меня", command=hello)
>>> btn.pack()
>>> привет
привет
привет
привет
привет
                                              Ln: 17  Col: 4
```

```
tk
нажми меня
```

В этом примере мы впервые использовали именованные аргументы, поэтому поговорим об этом подробнее.

Именованные аргументы

Именованные аргументы не отличаются от обычных, но чтобы определить, какое значение при вызове функции к какому аргументу относится, нужно использовать не очередность (первое значение — первый аргумент, второе — второй и так далее), а имена аргументов. Тогда порядок передачи значений роли уже не играет.

Некоторые функции принимают множество аргументов, причем некоторые из них указывать не обязательно. Именованные аргументы позволяют указать значения только для тех аргументов, которые мы хотим задать.

Person — человек
Width — ширина
Height — высота

Допустим, у нас есть функция `person`, принимающая два аргумента: `width` и `height`.

```
>>> def person(width, height):
        print('Моя ширина - %s, а высота - %s' % (width, height))
```

Обычно мы вызываем подобные функции так:

```
>>> person(4, 3)
Моя ширина - 4, а высота - 3
```

Именованные аргументы позволяют вызвать эту функцию, указывая для каждого значения имя соответствующего аргумента:

```
>>> person(height=3, width=4)
Моя ширина - 4, а высота - 3
```

Чем ближе мы будем знакомиться с модулем tkinter, тем полезнее будут для нас именованные аргументы.

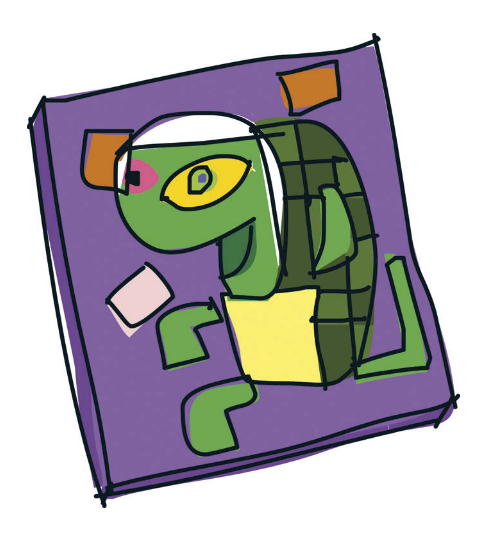
Создаем холст для рисования

Кнопки — полезные элементы управления. Однако, если мы хотим нарисовать что-то на экране, толку от них немного. Чтобы создать новое изображение, нужен другой элемент, а именно холст — объект класса Canvas, который тоже входит в модуль tkinter.

Canvas — холст

Для создания холста нужно указать его ширину (width) и высоту (height). В остальном код аналогичен созданию кнопки. Например:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
```



Как и в примере с кнопкой, при вводе строки tk = Tk() на экране появится окно. В последней строке кода мы вызываем функцию canvas.pack(), после чего холст примет размеры 500 пикселей в ширину и 500 в высоту, как указано в третьей строке кода.

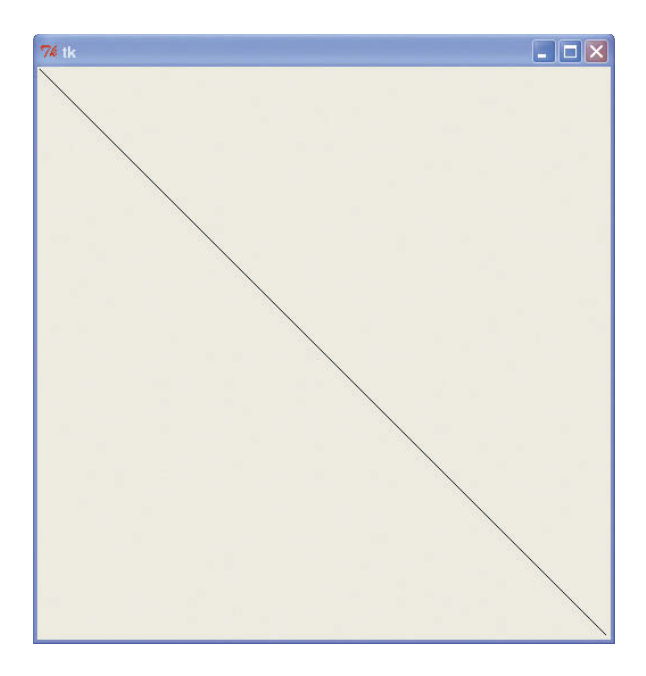
Аналогично примеру с кнопкой, функция pack размещает холст внутри окна. Без вызова этой функции содержимое окна не будет отображаться правильно.

Pack — упаковать, разместить

Рисование линий

В tkinter для рисования линий на холсте используются координаты в пикселях. Координаты — это два числа, которые определяют, насколько пиксель (точка) отстоит от левого края холста по горизонтали и от верхнего края холста по вертикали.

Поскольку наш холст занимает 500 пикселей в ширину и 500 в высоту, его правому нижнему углу соответствуют координаты (500, 500). Чтобы изобразить линию, показанную на рисунке ниже, нужно начать рисование с координат (0, 0) и закончить координатами (500, 500).



Create line — создать линию

Эти координаты нужно передать функции create_line, которая рисует линию на холсте. Вот так:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
>>> canvas.create_line(0, 0, 500, 500)
1
```

Функция create_line вернула число 1 (это идентификатор, и мы поговорим об этом позже). Чтобы нарисовать такую же линию с помощью черепашки, нам пришлось бы написать следующий код:

```
>>> import turtle
>>> turtle.setup(width=500, height=500)
>>> t = turtle.Pen()
>>> t.up()
>>> t.goto(-250, 250)
>>> t.down()
>>> t.goto(500, -500)
```

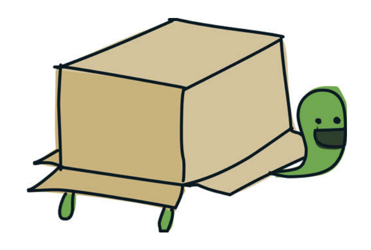
Go to — перейти к

Преимущества уже налицо — с tkinter код стал короче и проще.

Теперь познакомимся с функциями холста tkinter, с помощью которых можно изображать более интересные фигуры.

Рисование прямоугольников

Раньше мы рисовали прямоугольники, двигая черепашку вперед, поворачивая, затем снова двигая вперед, опять поворачивая и так далее. Размеры прямоугольника или квадрата зависели от того, на какие расстояния перемещалась черепашка.

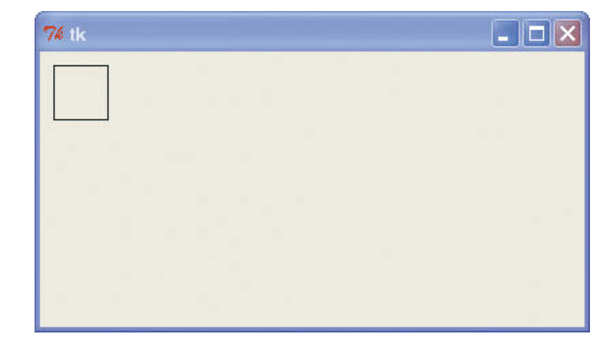


С помощью модуля tkinter нарисовать прямоугольник гораздо проще. Все, что вам нужно знать, — это координаты его углов. Вот пример (окна, оставшиеся от предыдущих примеров, можно закрыть):

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 50)
```

Create rectangle — создать прямоугольник

Мы создали холст размером 400 на 400 пикселей, а затем нарисовали в левом верхнем углу окна квадрат. Вот такой:



Аргументы, которые мы передали функции `canvas.create_rectangle` в последней строке кода, — это координаты левого верхнего и правого нижнего углов квадрата. Первые два значения соответствуют верхнему левому углу (10 пикселей вправо от левого края и 10 пикселей вниз от верхнего), а вторые два — правому нижнему углу (50 пикселей от левого края и 50 — от верхнего).

Назовем эти две координаты *x1*, *y1* и *x2*, *y2*. Чтобы вместо квадрата получился вытянутый прямоугольник, можно изменить координаты правого нижнего угла, увеличив значение *x2*. Вот так:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 300, 50)
```

В этом примере положению левого верхнего угла прямоугольника соответствуют координаты (10, 10), а положению правого нижнего угла — координаты (300, 50). Получается прямоугольник такой же высоты, что и квадрат в предыдущем примере, но значительно большей ширины.



Сделать из квадрата прямоугольник можно и по-другому — увеличив расстояние от верхнего края холста до правого нижнего угла (увеличив значение *y2*). Вот так:

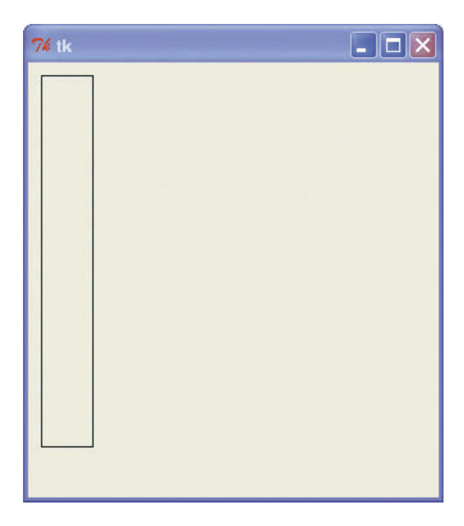
```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 300)
```

Вызывая функцию `create_rectangle`, мы говорим примерно следующее:

- отступить на 10 пикселей вправо от левого края холста;

- отступить на 10 пикселей вниз от верхнего края холста (это левый верхний угол прямоугольника);
- рисовать прямоугольник шириной 50 пикселей…
- …и высотой 300 пикселей.

Вот что должно получиться:



Рисуем множество прямоугольников

Заполним холст прямоугольниками разных размеров. Для этого импортируем модуль `random`, а затем создадим функцию, которая рисует прямоугольник, используя для координат углов случайные числа.

Random — случайный, произвольный

Воспользуемся функцией `randrange` из модуля `random`. Эта функция возвращает случайное число в диапазоне от 0 до значения, которое мы передадим ей в качестве аргумента (но не включая это значение). Например, вызов `randrange(10)` вернет число от 0 до 9, а вызов `randrange(100)` — число от 0 до 99 и так далее.

Randrange — от random range — произвольный диапазон

Используем `randrange` в функции рисования прямоугольника следующим образом. Откройте новое окно, выбрав в меню **File ▸ New File**, и введите такой код:

```
from tkinter import *
import random
```

```
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
def random_rectangle(width, height):
        x1 = random.randrange(width)
        y1 = random.randrange(height)
        x2 = x1 + random.randrange(width)
        y2 = y1 + random.randrange(height)
        canvas.create_rectangle(x1, y1, x2, y2)
```

Random rectangle — произвольный прямоугольник

Мы определили функцию `random_rectangle`, принимающую два аргумента: максимальную ширину (`width`) и максимальную высоту (`height`). Создали переменные `x1` и `y1` для левого верхнего угла прямоугольника, получив значение каждой из них из функции `randrange`, в которую передали ширину (для `x1`) и высоту (для `y1`). Вторая строка нашей функции означает: создать переменную `x1` и поместить в нее случайное число в диапазоне от 0 до аргумента `width`.

В следующих двух строках создаем переменные `x2` и `y2` для правого нижнего угла. Берем для этого координаты левого верхнего угла (`x1` и `y1` соответственно) и прибавляем к ним случайные числа. Третья строка нашей функции означает: создать переменную `x2` и поместить в нее сумму случайного числа и значения переменной `x1`.

Вызываем функцию `canvas.create_rectangle`, передавая ей значения `x1`, `y1`, `x2` и `y2`, чтобы изобразить прямоугольник на холсте.

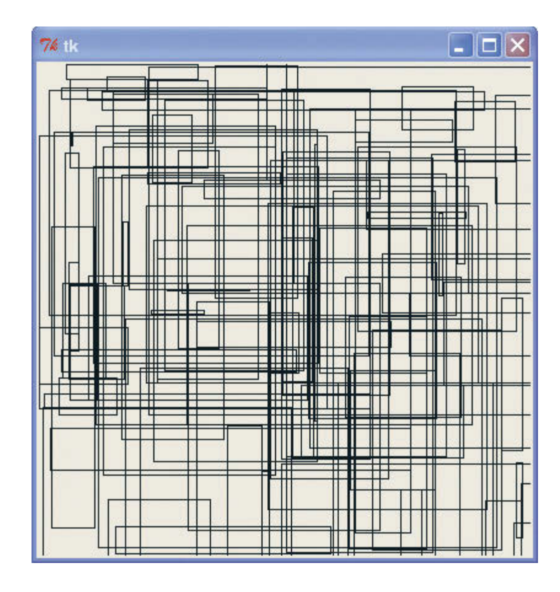
Проверим работу функции `random_rectangle`, передав ей ширину и высоту холста. Введите этот код после функции, которую мы только что создали:

```
random_rectangle(400, 400)
```

Сохраните код (выберите **File ▸ Save** и введите имя файла, например *randomrect.py*), а затем выберите **Run ▸ Run Module**. Убедившись, что функция `random_rectangle` работает, заполним экран прямоугольниками, многократно вызывая ее в цикле, скажем, 100 раз. Добавьте этот код после предыдущего, сохраните файл и снова запустите программу:

```
for x in range(0, 100):
    random_rectangle(400, 400)
```

Получится вот такая мешанина:



Рисование в цвете

Мы хотим использовать в рисунках различные цвета. Давайте изменим функцию `random_rectangle` так, чтобы она принимала цвет прямоугольника в качестве дополнительного аргумента с именем `fill_color`. Введите в новом окне следующий код и сохраните его под именем *colorrect.py*:

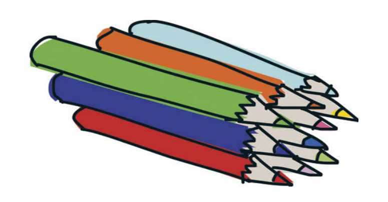
Fill color — цвет заливки

Colorrect — от color rectangle — цветной прямоугольник

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()

def random_rectangle(width, height, fill_color):
        x1 = random.randrange(width)
        y1 = random.randrange(height)
        x2 = random.randrange(x1 + random.randrange(width))
        y2 = random.randrange(y1 + random.randrange(height))
        canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color)
```

Теперь функция `create_rectangle` принимает аргумент `fill_color` со значением цвета, которым будет заполнен прямоугольник.



В нашу функцию можно передавать названия разных цветов, как показано ниже (для холста 400 на 400 пикселей), и получится набор разноцветных прямоугольников. Вводя код этого примера, имеет смысл воспользоваться копированием и вставкой текста, чтобы не набирать по многу раз однотипные команды. Для этого выделите фрагмент текста для копирования, нажмите **Ctrl-C**, кликните мышкой по пустой строке и вставьте ранее скопированный текст, нажав **Ctrl-V**. Добавьте этот код в окно с файлом *colorrect.py* следом за функцией `random_rectangle`:

Green — зеленый
Red — красный
Blue — синий
Orange — оранжевый
Yellow — желтый
Pink — розовый
Purple — сиреневый
Violet — фиолетовый
Magenta — пурпурный
Cyan — *здесь* голубой

```
random_rectangle(400, 400, 'green')
random_rectangle(400, 400, 'red')
random_rectangle(400, 400, 'blue')
random_rectangle(400, 400, 'orange')
random_rectangle(400, 400, 'yellow')
random_rectangle(400, 400, 'pink')
random_rectangle(400, 400, 'purple')
random_rectangle(400, 400, 'violet')
random_rectangle(400, 400, 'magenta')
random_rectangle(400, 400, 'cyan')
```

Вводя некоторые из названий цветов на английском, вы можете получить вместо ожидаемого результата ошибку — это зависит от того, какую операционную систему вы используете (Windows, Mac OS X или Linux).

Как быть с цветами, которые не соответствуют ни одному из названий? Вспомните, как в главе 11 мы задавали цвет пера черепашки, используя насыщенность красного, зеленого и синего цветов в процентах. Задать насыщенность основных цветов (красного, зеленого и синего) для создания смешанного цвета в `tkinter` несколько сложнее. Сейчас мы с этим разберемся.

Используя модуль `turtle`, мы задавали золотой цвет, смешивая 90 процентов красного, 75 процентов зеленого и не добавляя синий. В `tkinter` такой же цвет можно задать следующим образом:

```
random_rectangle(400, 400, '#ffd800')
```

Символ решетки (#) перед значением ffd800 означает, что это число в *шестнадцатеричной* системе счисления. Шестнадцатеричная система — это способ представления чисел, который широко используется в программировании. В этой системе счисления основание — 16 (цифры от 0 до 9 и буквы от A до F), а не 10 (цифры от 0 до 9), как в привычной нам десятичной системе. Десятичное число можно преобразовать в шестнадцатеричное с помощью *метки формата* `%x` внутри строки (см. раздел

«Переменные внутри строк» на стр. 36). Например, чтобы преобразовать число 15 из десятичной системы в шестнадцатеричную, введите:

```
>>> print('%x' % 15)
f
```

Если требуется выводить не менее двух знаков числа, метку формата можно слегка изменить:

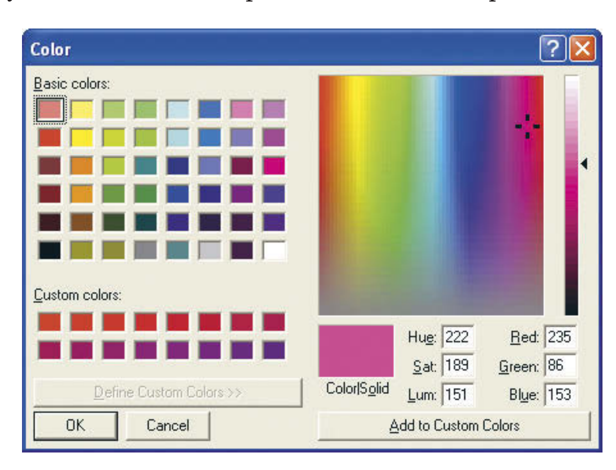
```
>>> print('%02x' % 15)
0f
```

В модуле `tkinter` есть инструмент, с помощью которого несложно узнать шестнадцатеричное значение цвета. Добавьте следующий код в файл *colorrect.py* (вызовы функции `random_rectangle` из предыдущего примера можно удалить).

```
from tkinter import *
colorchooser.askcolor()
```

Color chooser — средство выбора цвета
Ask color — спросить цвет

При запуске этого кода откроется диалог выбора цвета:



Выбрав цвет и кликнув ОК, вы увидите в окне оболочки кортеж, содержащий другой кортеж с тремя числами внутри, а также строку:

```
>>> colorchooser.askcolor()
((235.91796875, 86.3359375, 153.59765625), '#eb5699')
```

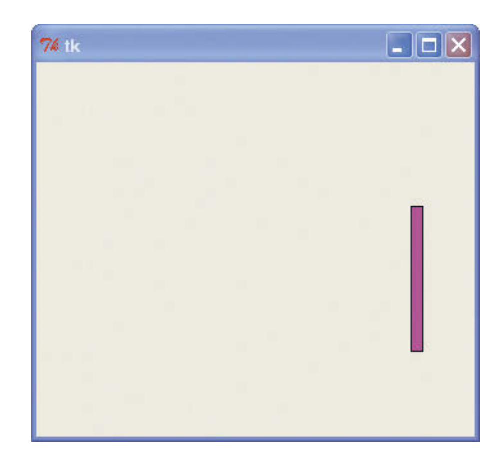
Три числа — это насыщенность красной, зеленой и синей составляющих цвета. В tkinter насыщенность основных цветов обозначается числами от 0 до 255 (а не от 0 до 1, как в модуле turtle). Строка в кортеже содержит шестнадцатеричное представление этих трех чисел.

Можно скопировать и вставить это значение в код, а можно сохранить кортеж в переменной и обращаться к шестнадцатеричному значению по его индексу.

Посмотрим, как это работает на примере функции random_rectangle:

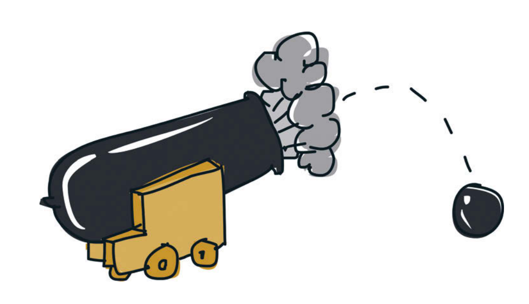
```
>>> c = colorchooser.askcolor()
>>> random_rectangle(400, 400, c[1])
```

Вот результат:



Рисование дуг

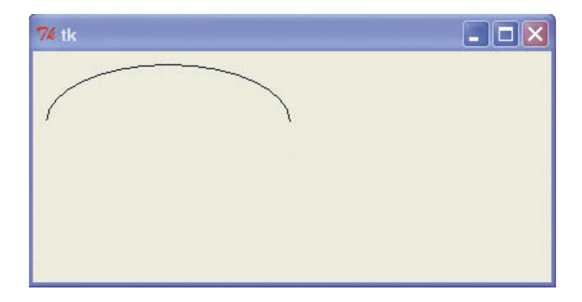
Дуга — это сегмент окружности или эллипса. Чтобы изобразить дугу с помощью tkinter, нужно вызвать функцию create_arc, передав ей координаты углов прямоугольника, в который будет вписана эта дуга. Вот так:



Create arc — создать дугу

Extent — здесь «размах» или «разворот»
Style — стиль

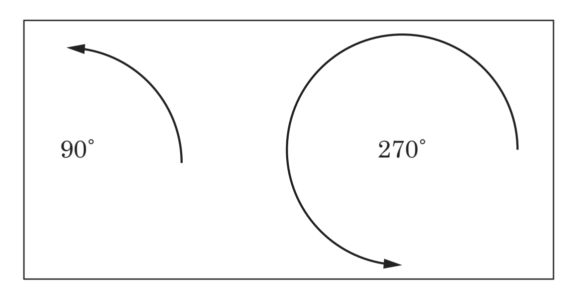
```
canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```



Если вы уже закрыли все окна `tkinter` или перезапускали IDLE, нужно снова импортировать `tkinter` и создать холст, введя такой код:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

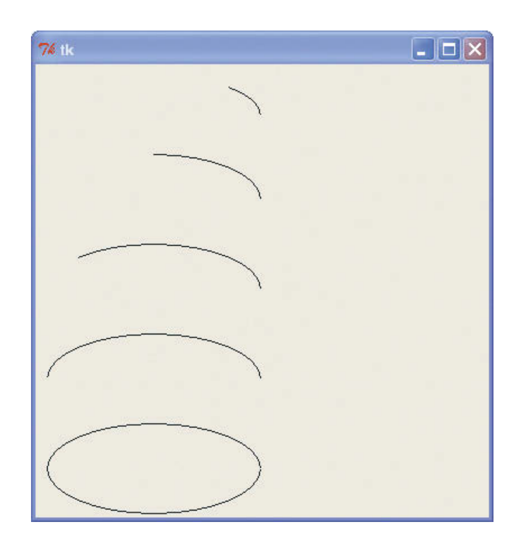
Левый верхний угол описывающего прямоугольника будет помещен в точку с координатами (10, 10), что соответствует расстоянию в 10 пикселей от верхнего края холста и 10 пикселей от его левого края, а правый нижний угол — в точку с координатами (200, 100). Далее указан аргумент `extent`, в котором передается угол разворота дуги в градусах. Как мы знаем из главы 4, с помощью градусов можно измерять угол. Вот две дуги, разворот первой равен 45 градусам, а второй — 270 градусам.



Нарисуем несколько дуг — одну под другой. Так будет видно, что происходит при передаче в функцию `create_arc` разных значений угла.

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
```

```
>>> canvas.create_arc(10, 10, 200, 80, extent=45, style=ARC)
>>> canvas.create_arc(10, 80, 200, 160, extent=90, style=ARC)
>>> canvas.create_arc(10, 160, 200, 240, extent=135, style=ARC)
>>> canvas.create_arc(10, 240, 200, 320, extent=180, style=ARC)
>>> canvas.create_arc(10, 320, 200, 400, extent=359, style=ARC)
```



! *В качестве полного оборота используем угол 359 градусов, а не 360, поскольку* `tkinter` *считает, что угол 360 градусов равен углу 0 градусов, и ничего не нарисует.*

Рисование многоугольников

Многоугольник — это любая фигура с тремя или более углами, контур которой замкнут. Многоугольники бывают правильными (у которых все стороны и углы равны) и неправильными (с разными величинами сторон и углов).

Чтобы нарисовать многоугольник с помощью `tkinter`, нужно указать координаты точек, соответствующих всем его углам. Изобразить треугольник можно так:

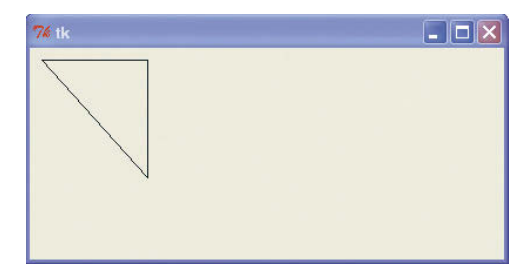
Polygon — многоугольник

Outline — контур

Black — черный

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 100, 10, 100, 110, fill="",
outline="black")
```

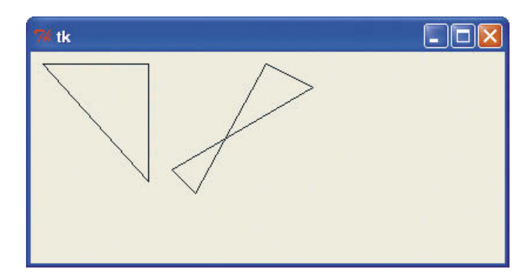
Получится треугольник, контур которого соединяет точку с координатами (10, 10) с точкой (100, 10) и точкой (100, 110):



Добавим еще один неправильный многоугольник (с неравными сторонами и неравными углами), введя следующий код:

```
canvas.create_polygon(200, 10, 240, 30, 120, 100, 140, 120,
fill="", outline="black")
```

Получится фигура, контур которой соединяет точку с координатами (200, 10) с точкой (240, 30), точкой (120, 100) и точкой (140, 120); `tkinter` автоматически замкнет контур, проведя линию обратно к первой точке, и получится вот что:



Отображение текста

Помимо рисования фигур на холсте, можно печатать сообщения с помощью функции `create_text`, которая принимает две координаты (*x*- и *y*-позиции надписи), а также именованный аргумент со строкой текста. В следующем примере мы создаем холст так же, как и прежде, а затем печатаем строку в позиции (150, 100). Сохраните этот код в файле *text.py*.

Create text — создать текст

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_text(150, 100, text='Был один человек из Тулузы,')
```

Функция create_text может принимать и другие полезные аргументы, например цвет надписи. В следующей строке кода вызываем create_text, передавая координаты (130, 120), сообщение для печати, а также цвет (красный):

```
canvas.create_text(130, 120, text='Что сидел на огромном арбузе.',
fill='red')
```

Font — шрифт

В аргументе font можно передать кортеж, который содержит название и размер шрифта выводимого на экран сообщения. Например, для шрифта Times размером 20 пунктов кортеж будет таким: ('Times', 20).

!

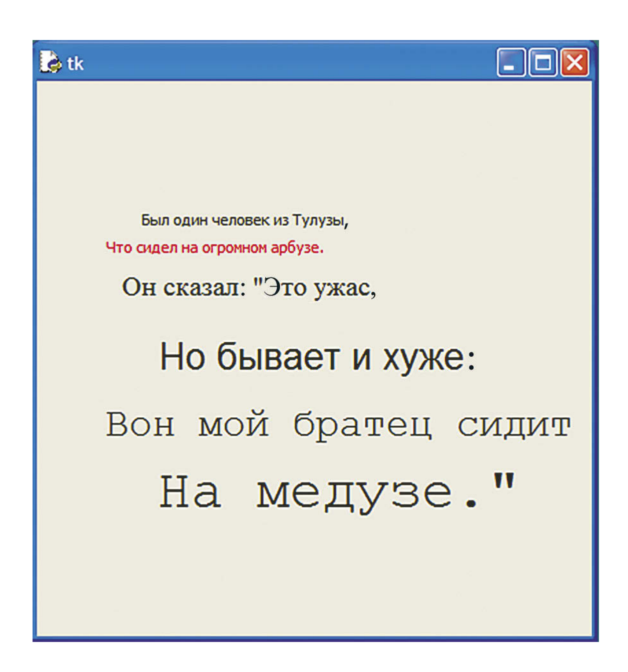
Размер шрифта здесь указывается в пунктах, а не в пикселях. Используемый в компьютерной графике пункт соответствует 0,3527 мм. В tkinter можно указать размер шрифта и в пикселях. Для этого используются отрицательные значения. Например, для шрифта Times размером 20 пикселей: ('Times', -20).



В следующем примере строки текста выводятся шрифтом Times размера 15, шрифтом Helvetica размера 20 и шрифтом Courier размера 22, а затем размера 30.

```
canvas.create_text(150, 150, text='Он сказал: "Это ужас,',
font=('Times', 15))
canvas.create_text(200, 200, text='Но бывает и хуже:',
font=('Helvetica', 20))
canvas.create_text(220, 250, text='Вон мой братец сидит',
font=('Courier', 22))
canvas.create_text(220, 300, text='На медузе".', font=('Courier',
30))
```

Вот результат всех этих вызовов create_text с разными шрифтами разных размеров:

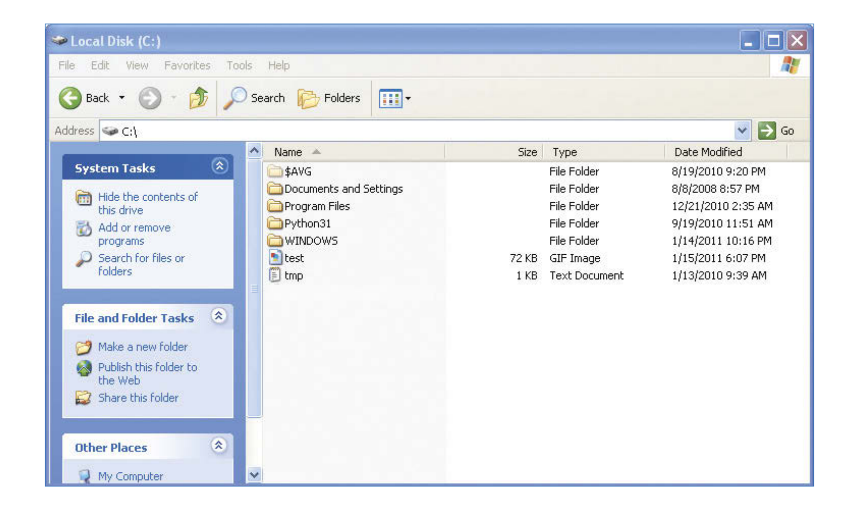


Вывод изображений

Чтобы с помощью `tkinter` вывести на холст изображение, его сначала нужно загрузить, а затем вызвать функцию `create_image`.

Create image — создать изображение

Файл с изображением должен находиться в папке, к которой у Python есть доступ. Допустим, изображение *test.gif* находится в папке *C:*\\, то есть корневой директории диска *C:* (хотя его можно положить и в другое место).



Если вы используете Mac OS или Linux, можете поместить изображение в свою домашнюю директорию. Если в корень диска *C:* не получается скопировать файл, можете поместить его на рабочий стол.

! *Средствами* `tkinter` *можно загружать только изображения формата GIF (файлы с расширением .gif). Чтобы вывести изображение другого типа, скажем, PNG (.png) или JPG (.jpg), придется воспользоваться другим модулем, например Python Imaging Library (http://www.pythonware.com/products/pil/).*

Вывести изображение *test.gif* на экран можно так:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
my_image = PhotoImage(file='c:\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=my_image)
```

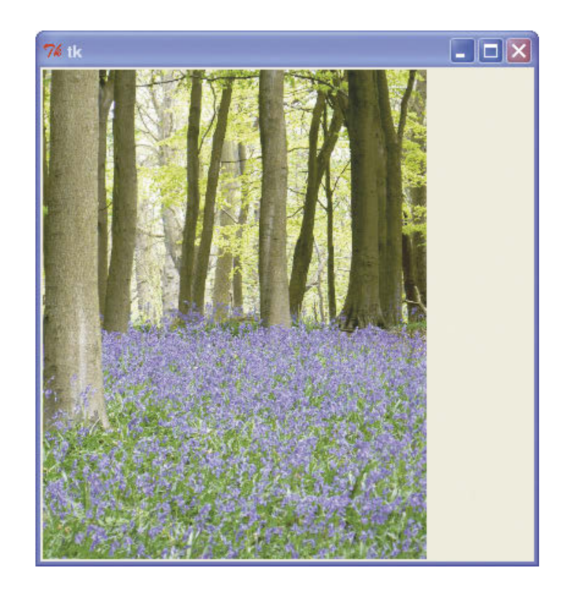
В первых четырех строках мы создали холст так же, как в предыдущих примерах. В пятой строке загрузили изображение в переменную `my_image`. Если ваше изображение находится на рабочем столе, нужно передать в функцию `PhotoImage` другой путь, например такой:

Photo Image — фотоизображение

```
my_image = PhotoImage(file='C:\\Users\\Joe Smith\\Desktop\\test.gif')
```

NW — сокращение от northwest — северо-запад

Когда изображение загружено, команда `canvas.create_image(0, 0, anchor=NW, image=my_image)` выводит его на экран. Координаты `(0, 0)` определяют позицию изображения, аргумент `anchor=NW` означает, что изображение должно выводиться с левого верхнего угла (иначе за отправную точку считался бы центр изображения). Последний именованный аргумент — `image` — указывает на переменную с загруженным изображением. Вот что должно получиться:



Создание простой анимации

До сих пор мы занимались только статичными, неизменяющимися рисунками. Пора переходить к анимации.

Модуль `tkinter` не ориентирован на создание анимации, но что-нибудь простое с его помощью сделать можно. Например, следующий код рисует заполненный треугольник, который движется по экрану слева направо (не забудьте выбрать **File ▸ New File**, сохранить код на диск и затем запустить его, выбрав **Run ▸ Run Module**):

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=200)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
        canvas.move(1, 5, 0)
        tk.update()
        time.sleep(0.05)
```

После запуска вы увидите, как треугольник движется по экрану вправо и останавливается:



Как это работает? В первых трех строках кода после импортирования `tkinter` мы, как обычно, создаем и настраиваем холст. В четвертой строке создаем треугольник с помощью вызова `create_polygon`:

```
canvas.create_polygon(10, 10, 10, 60, 50, 35)
```

Если ввести эту строку в оболочке IDLE, на экране появится число — идентификатор нашего треугольника, который можно использовать для обращения к созданной фигуре.



Задаем простой цикл `for`, который считает от 0 до 59 (команда `for x in range(0, 60)`). Выполняющийся в этом цикле блок кода служит для перемещения треугольника по экрану. Функция `canvas.move` двигает любой нарисованный ранее графический объект, изменяя значения его *x*- и *y*-координат на указанные величины смещения. В частности, команда `canvas.move(1, 5, 0)` сдвинет объект, идентификатор которого равен 1 (это наш треугольник), на 5 пикселей вправо и 0 пикселей вниз. А для возвращения треугольника на прежнее место подойдет команда `canvas.move(1, -5, 0)`.

Функция `tk.update()` служит для принудительного обновления (перерисовки) изображения на экране. Без этой команды `tkinter` обновил бы экран лишь после окончания цикла и вместо плавной анимации мы бы увидели треугольник сразу в конечной точке движения. А последняя команда в теле цикла, `time.sleep(0.05)`, задает 0,05-секундную паузу.

Предположим, мы хотим, чтобы треугольник двигался по диагонали вправо и вниз. Для этого можно изменить код, использовав команду `move(1, 5, 5)`. Закройте окно с холстом, создайте новый файл (**File ▸ New File**) и введите:

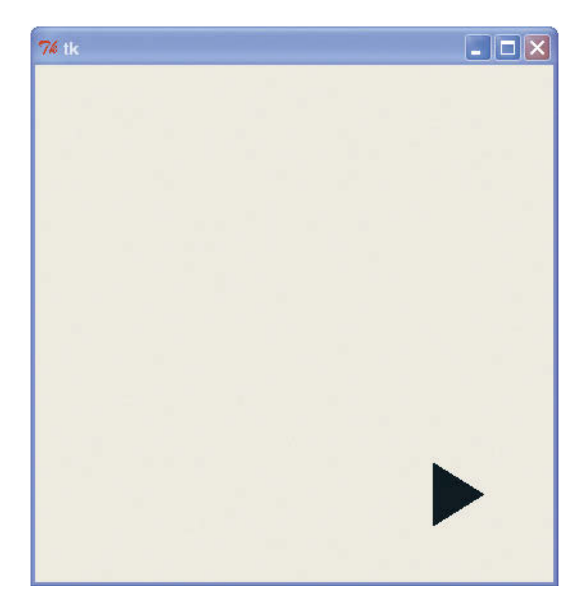
```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
        canvas.move(1, 5, 5)
        tk.update()
        time.sleep(0.05)
```

Здесь два отличия от предыдущего примера с треугольником:

1. Задаем высоту холста 400 пикселей (а не 200) с помощью команды `canvas = Canvas(tk, width=400, height=400)`.

2. Прибавляем 5 не только к *x*-, но и к *y*-координате треугольника, вызывая `canvas.move(1, 5, 5)`.

Если вы сохраните программу, запустите ее и дождетесь окончания работы цикла, треугольник окажется здесь:



Чтобы переместить треугольник по диагонали обратно, в начальную позицию, можно использовать такой же цикл со смещениями –5, –5 (добавьте этот код в конец файла):

```
for x in range(0, 60):
        canvas.move(1, -5, -5)
        tk.update()
        time.sleep(0.05)
```

Реакция объектов на события

Можно сделать так, чтобы треугольник реагировал на нажатия клавиш. Для этого служит так называемая *привязка к событиям*. *События* — то, что происходит во время работы программы, например при передвижении мышки, нажатии клавиш или работе с окнами. Можно попросить `tkinter` следить за такими событиями и реагировать на них.

Для обработки события нужно создать функцию, а затем сообщить `tkinter`, что ее следует привязать к определенному событию, иначе говоря, что она должна вызываться, когда это событие произойдет.

Пусть треугольник будет двигаться при нажатии клавиши ENTER. Создадим для этого такую функцию:

```
def movetriangle(event):
canvas.move(1, 5, 0)
```

Move triangle — подвинуть треугольник

Event — событие

Наша функция принимает единственный аргумент (event), в котором tkinter передает информацию о событии. Теперь нужно, чтобы tkinter вызывал эту функцию при нажатии ENTER. Используем для этого функцию bind_all. Весь код будет выглядеть так:

Bind all — *здесь* назначить для всех (элементов)

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
    canvas.move(1, 5, 0)
canvas.bind_all('<KeyPress-Return>', movetriangle)
```

Первый аргумент функции bind_all описывает событие, за которым нужно наблюдать. В данном случае оно называется <KeyPress-Return>, что соответствует нажатию клавиши **Enter** или **Return**. С помощью второго аргумента мы указываем, что при возникновении события нужно вызывать функцию movetriangle. Запустите этот код, кликните мышкой по холсту и понажимайте ENTER на клавиатуре.



Если мы хотим менять направление движения треугольника в зависимости от нажатий на разные клавиши, скажем, на клавиши-стрелки, нужно изменить код функции movetriangle:

```
def movetriangle(event):
        if event.keysym == 'Up':
              canvas.move(1, 0, -3)
        elif event.keysym == 'Down':
              canvas.move(1, 0, 3)
        elif event.keysym == 'Left':
              canvas.move(1, -3, 0)
        else:
              canvas.move(1, 3, 0)
```

Объект event, который передается в функцию movetriangle, содержит набор свойств (переменных объекта). В одном из них, keysym, хранится значение, соответствующее нажатой клавише. Строка if event.keysym == 'Up': означает, что, если в свойстве keysym находится значение 'Up', нужно выполнить следующую строку кода: вызвать canvas.move с аргументами (1, 0, -3). Если же в keysym содержится

Keysym — *здесь* имя клавиши

'Down' (elif event.keysym == 'Down':), нужно вызвать canvas.move с аргументами (1, 0, 3), и так далее.

Напоминаю: первый аргумент move — это идентификатор графического объекта на холсте, второй — значение, на которое следует изменить координату *x* (позицию по горизонтали), а третий — значение, на которое нужно изменить координату *y* (позицию по вертикали).

Осталось сообщить tkinter, что функцию movetriangle нужно использовать для обработки четырех событий: нажатий клавиш-стрелок «вверх», «вниз», «влево» и «вправо». Вот как будет выглядеть код (его будет гораздо проще вводить, создав новый файл (**File ▸ New File**)). Перед запуском сохраните файл — например, под именем *movingtriangle.py*.

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
❶        if event.keysym == 'Up':
❷              canvas.move(1, 0, -3)
❸        elif event.keysym == 'Down':
❹              canvas.move(1, 0, 3)
❺        elif event.keysym == 'Left':
❻              canvas.move(1, -3, 0)
❼        else:
❽              canvas.move(1, 3, 0)
canvas.bind_all('<KeyPress-Up>', movetriangle)
canvas.bind_all('<KeyPress-Down>', movetriangle)
canvas.bind_all('<KeyPress-Left>', movetriangle)
canvas.bind_all('<KeyPress-Right>', movetriangle)
```

В функции movetriangle в строке ❶ проверяем, содержит ли переменная keysym значение 'Up'. Если содержит, перемещаем треугольник вверх, вызывая в строке ❷ функцию move с аргументами 1, 0, -3. Первый аргумент move — идентификатор треугольника, второй — величина сдвига вправо (нам не нужно горизонтальное смещение, поэтому указываем значение 0), а третий — величина сдвига вниз (–3 пикселя).

В строке ❸ проверяем, не содержит ли keysym значения 'Down', и если содержит, в строке ❹ сдвигаем треугольник вниз. В строке ❺ идет последняя проверка — на значение 'Left'. Если это подтверждается, в строке ❻ сдвигаем треугольник влево (–3 пикселя). Если же ни одна из проверок не сработала, в строке ❼ выполнится вариант else и в строке ❽ треугольник сдвинется вправо.

При нажатии любой из клавиш-стрелок треугольник будет перемещаться в соответствующем направлении.

Для чего еще нужен идентификатор

Функции холста, имена которых начинаются с create_ (создать), такие как create_polygon или create_rectangle, при каждом вызове возвращают число-идентификатор, который можно использовать с другими функциями холста, например функцией move (двигать), которую мы использовали ранее:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> canvas.move(1, 5, 0)
```

В этом коде есть одна проблема — create_polygon может вернуть не 1, а иное значение. Например, если до этого мы уже создавали какие-то фигуры, она может вернуть число 2, 3 или даже 100 (в зависимости от количества созданных фигур). Однако, если мы доработаем программу так, чтобы возвращаемое значение сохранялось в переменной и она использовалась как идентификатор (вместо числа 1), код будет работать независимо от того, какое именно значение было возвращено:

```
>>> mytriangle = canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> canvas.move(mytriangle, 5, 0)
```

Функция move позволяет перемещать графические объекты по экрану с помощью их идентификаторов. Но есть и другие функции холста, которые также способны менять созданные ранее объекты. Например, функцию itemconfig можно использовать для изменения некоторых свойств фигуры, таких как цвет заполнения и цвет обводки.

Item config — от item configuration — конфигурация элемента

Положим, у нас есть красный треугольник, созданный таким образом:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> mytriangle = canvas.create_polygon(10, 10, 10, 60, 50, 35,
fill='red')
```

Можно изменить цвет этого треугольника, вызвав функцию itemconfig с его идентификатором в качестве первого аргумента.

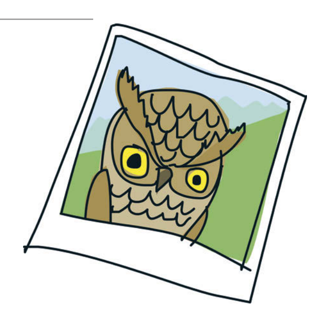
Следующий код означает: изменить цвет заполнения объекта, идентификатор которого находится в переменной `mytriangle`, на синий (`blue`).

```
>>> canvas.itemconfig(mytriangle, fill='blue')
```

Можно поменять цвет обводки треугольника на, допустим, красный цвет (опять же, передавая идентификатор первым аргументом):

```
>>> canvas.itemconfig(mytriangle, outline='red')
```

Скоро мы научимся менять и другие свойства графических объектов — например делать их невидимыми, а затем снова видимыми. Когда в следующей главе мы перейдем к написанию игр, вы убедитесь, что возможность менять уже созданные рисунки очень полезна.



Что мы узнали

В этой главе с помощью модуля `tkinter` мы рисовали на холсте простые геометрические фигуры, выводили изображения, а также показывали несложную анимацию. Узнали, как, используя привязку к событиям, «научить» рисунки на холсте реагировать на нажатия клавиш — это еще пригодится нам для создания игры. Выяснили, что функции создания графических объектов в `tkinter` возвращают идентификаторы, чтобы эти объекты можно было изменять, например перемещать их по экрану или менять цвет.

Упражнения

Попрактикуйтесь в работе с модулем `tkinter` и несложной анимацией.

#1. Заполните экран треугольниками

Напишите программу, которая с помощью `tkinter` заполняет экран треугольниками. Затем модифицируйте код, чтобы треугольники были раскрашены (заполнены) различными цветами.

#2. Движущийся треугольник

Доработайте код движущегося треугольника (см. «Создание простой анимации» на стр. 181), чтобы треугольник двигался вправо, вниз, влево и вверх, вернувшись в итоге в первоначальную позицию.

#3. Движущаяся фотография

С помощью `tkinter` отобразите на экране свою фотографию. Не забывайте, что изображение должно быть в формате GIF! А теперь сделайте так, чтобы фотография перемещалась по экрану.





ЧАСТЬ II

Пишем игру «Прыг-скок!»

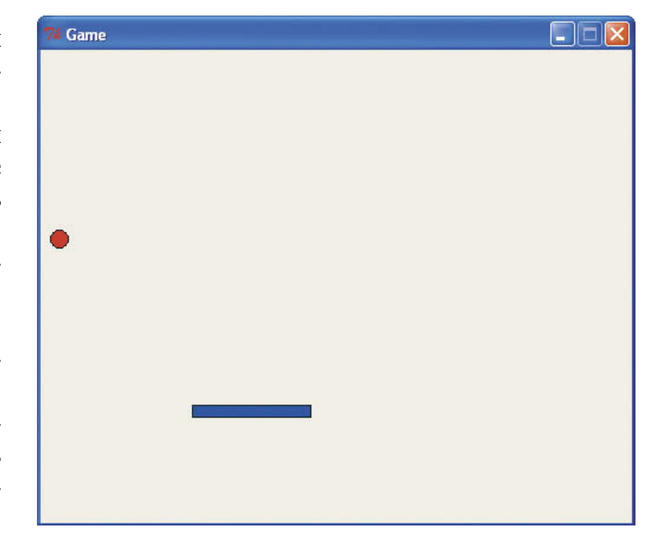
13

НАША ПЕРВАЯ ИГРА: «ПРЫГ-СКОК!»

В предыдущих главах мы изучили основы программирования. Узнали, как хранить данные в переменных, работать с конструкцией `if` с условиями и многократно выполнять код с помощью циклов. Научились создавать функции, чтобы использовать код повторно, и разделять программу на более простые и понятные фрагменты с помощью классов. Познакомились с основами программирования графики с использованием модулей `turtle` и `tkinter`. Пришло время применить все эти знания на практике, написав игру.

Прыгающий мяч

Разработаем игру с прыгающим мячом и ракеткой. Мяч будет летать по экрану, а игрок — отбивать его ракеткой. Если мяч коснется нижней границы экрана, игра завершится. На рисунке справа показано, как будет выглядеть законченная игра.



Хотя на первый взгляд игра довольно проста, ее код будет сложнее, чем все, что мы писали до сих пор, поскольку программа должна выполнять множество разных действий. К примеру, нужно анимировать ракетку и мяч, а также обрабатывать столкновения мяча с ракеткой и границами игрового поля.

В этой главе мы приступим к созданию игры, начав с холста и прыгающего мяча. А в следующей главе добавим ракетку, и получится законченная игра.

Создаем игровой холст

Первым делом откроем новый файл в оболочке Python (выберите **File ▸ New File**). Затем импортируем `tkinter` и создадим холст для рисования:

Resizable — изменяемый размер

Attributes — атрибуты

```
from tkinter import *
import random
import time
tk = Tk()
tk.title("Игра")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0,
highlightthickness=0)
canvas.pack()
tk.update()
```

Этот код несколько отличается от кода предыдущих примеров. Сначала помимо `tkinter` импортируем модули `time` и `random`, (`import random` и `import time`) — они нам понадобятся немного позже.

Вызовом `tk.title("Игра")` задаем заголовок игрового окна (для этого и служит функция `title` ранее созданного объекта `tk`). Вызываем функцию `resizable`, чтобы сделать размер окна фиксированным. Аргументы `0, 0` означают: размер окна должен быть неизменным как по горизонтали, так и по вертикали. Вызываем функцию `wm_attributes`, указывая, что окно с холстом нужно разместить поверх всех остальных окон (`"-topmost"`).

Обратите внимание, что, создавая холст, мы передали больше именованных аргументов, чем в предыдущих примерах. Например, аргументы `bd=0` и `highlightthickness=0`



нужны для того, чтобы вокруг холста не было рамки. Так наша игра будет лучше выглядеть.

В результате вызова `canvas.pack()` холст изменит размер в соответствии со значениями ширины и высоты, указанными в предыдущей строке кода. Команда `tk.update()` подготавливает `tkinter` к игровой анимации. Без вызова `update` программа не будет работать так, как задумано.

Не забывайте сохранять код по мере его написания. При первом сохранении дайте файлу имя, например *paddleball.py*.

Paddle ball — *букв.* ракетка мяч

Создаем класс для мяча

Теперь создадим класс для мяча и первым делом добавим в него код отрисовки мяча на холсте. Вот что нам предстоит сделать:

- создать класс под названием `Ball`, принимающий в качестве аргументов функции `__init__` холст и цвет мяча;
- сохранить в свойстве объекта холст, чтобы в дальнейшем рисовать на нем мяч;
- Изобразить на холсте круг, заполненный переданным в аргументе цветом;
- сохранить идентификатор, который вернет нам функция рисования круга, поскольку с его помощью мы будем перемещать мяч по экрану;
- переместить нарисованный круг в центр холста.

Ball — мяч

Этот код нужно добавить в начало файла, после строки `import time`:

```
   from tkinter import *
   import random
   import time

❶  class Ball:
❷      def __init__(self, canvas, color):
❸          self.canvas = canvas
❹          self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
❺          self.canvas.move(self.id, 245, 100)

       def draw(self):
           pass
```

Oval — овал, эллипс



В строке ❶ даем классу имя Ball. В строке ❷ создаем функцию инициализации __init__ (см. главу 8), которая принимает в качестве аргументов холст и цвет. В строке ❸ сохраняем аргумент canvas в свойстве с таким же именем.

В строке ❹ вызываем функцию create_oval для рисования круга, передавая ей пять аргументов: *x*- и *y*-координаты левого верхнего угла описывающего прямоугольника (10 и 10), *x*- и *y*-координаты его правого нижнего угла (25 и 25) и цвет заполнения.

Функция create_oval возвращает идентификатор нарисованной фигуры, который мы сохраняем в свойстве id. В строке ❺ перемещаем круг приблизительно в центр холста (позиция 245, 100), передав для этого в функцию move сохраненный ранее идентификатор фигуры (свойство id).

В двух последних строках определяем функцию draw (def draw(self)), указывая вместо тела функции ключевое слово pass (пока функция draw ничего не делает, но скоро мы это исправим).

Итак, у нас есть класс Ball, и теперь нужно создать объект этого класса (вспомните: класс лишь описывает, что нужно делать, а объект выполняет конкретные действия). Чтобы создать объект — мяч красного цвета, — добавьте в конец программы следующий код:

```
ball = Ball(canvas, 'red')
```

Если теперь запустить программу, выбрав **Run ▸ Run Module**, холст на мгновение появится и тут же исчезнет. Чтобы окно не закрывалось, нужно добавить в программу цикл игровой анимации, который называют *главным циклом* игры.

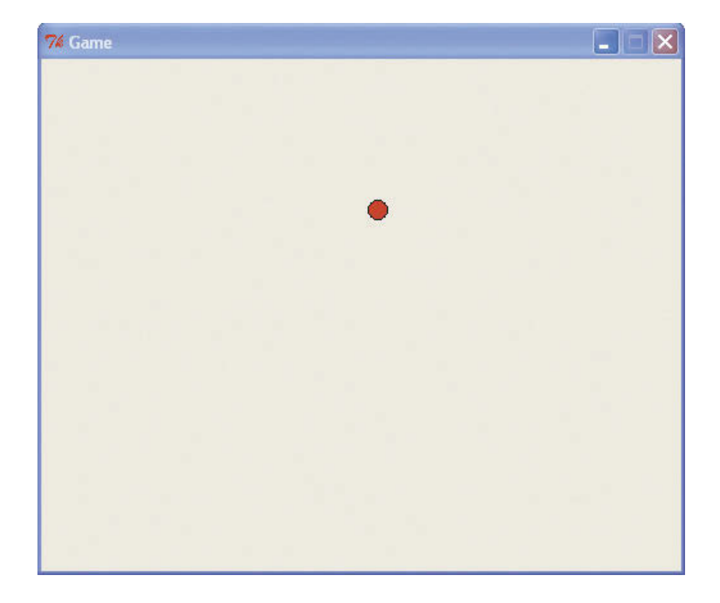
Главный цикл — центральный элемент программы, который управляет большей частью действий. Поначалу наш главный цикл будет только перерисовывать экран. Этот цикл бесконечный (по крайней мере он будет выполняться до тех пор, пока мы не закроем окно), и при каждом его повторе мы будем давать команду перерисовать экран и делать паузу на одну сотую долю секунды. Добавьте этот код в конец программы:

Tasks — задачи

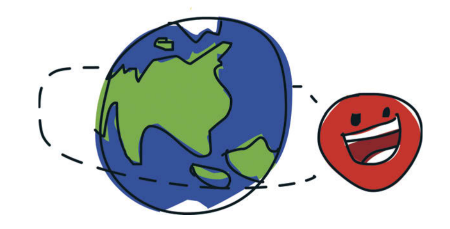
```
ball = Ball(canvas, 'red')

while 1:
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Если теперь запустить наш код, примерно в середине холста должен появиться мяч:



Добавим движение



Класс `Ball` подготовлен, теперь анимируем мяч. Наша задача — сделать так, чтобы мяч двигался и отскакивал от препятствий, меняя направление.

Перемещение мяча

Чтобы мяч двигался, первым делом изменим функцию `draw` таким образом:

```
class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)

    def draw(self):
        self.canvas.move(self.id, 0, -1)
```

Поскольку в функции `__init__` мы сохранили холст в свойстве `canvas`, теперь для вызова функции `move` можно обращаться к холсту через `self.canvas`.

Передаем функции `move` три аргумента: сохраненный ранее идентификатор круга (мяча), а также числа 0 и –1. Здесь 0 означает, что перемещения по горизонтали быть не должно, а –1 — что мяч должен переместиться на 1 пиксель вверх.

Мы внесли это небольшое изменение, чтобы посмотреть, как перемещается мяч. В процессе создания программы желательно почаще проверять ее части. Только представьте, что мы написали всю программу и вдруг обнаружили, что она не работает. Будет нелегко понять, в какую именно часть кода вкралась ошибка.

В главный цикл игры нужно внести еще одно изменение. Добавим в тело цикла `while` (это и есть главный цикл) вызов функции объекта-мяча `draw`. Вот так:

```
while 1:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Если теперь запустить код, мяч начнет двигаться вверх и вскоре исчезнет за границей окна, поскольку главный цикл быстро обновляет экран, постоянно перерисовывая его содержимое с помощью команд `update_idletasks` и `update`.

Sleep — *здесь* задержка

Команда `time.sleep` — это вызов функции `sleep` из модуля `time`. Она приостанавливает выполнение кода, в нашем случае — на сотую долю секунды (0.01). Это нужно для того, чтобы программа работала не слишком быстро, иначе мяч исчезнет прежде, чем мы успеем его разглядеть.

Итак, сейчас наш главный цикл немного сдвигает мяч, перерисовывает экран (после чего новое положение мяча становится видимым), делает небольшую паузу и повторяет эти действия снова и снова.

! *При закрытии окна игры в окне оболочки могут возникать сообщения об ошибках. Дело в том, что когда окно закрыто, действия с ним и его содержимым (которые выполняются в цикле) невозможны. На это и «жалуется» Python, выдавая ошибку.*

Сейчас код игры должен выглядеть так:

```
from tkinter import *
import random
import time
```

```
class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)

    def draw(self):
        self.canvas.move(self.id, 0, -1)

tk = Tk()
tk.title("Игра")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0,
highlightthickness=0)
canvas.pack()
tk.update()

ball = Ball(canvas, 'red')

while 1:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Отскоки мяча

Мяч, который улетает за границу холста, не очень подходит для этой игры, поэтому давайте «научим» его отскакивать от границ. Создадим в функции `__init__` класса `Ball` еще несколько свойств. Вот так:

```
def __init__(self, canvas, color):
    self.canvas = canvas
    self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
    self.canvas.move(self.id, 245, 100)
    self.x = 0
    self.y = -1
    self.canvas_height = self.canvas.winfo_height()
```

Мы добавили еще три строки кода. Команда `self.x = 0` задает свойству объекта с именем `x` значение 0, а `self.y = -1` задает свойству `y` значение –1. Сохраняем в свойстве `canvas_height` значение, полученное из функции холста `winfo_height`. Эта функция возвращает текущую высоту холста.

Снова изменим функцию `draw`:

```
def draw(self):
❶     self.canvas.move(self.id, self.x, self.y)
❷     pos = self.canvas.coords(self.id)
❸     if pos[1] <= 0:
          self.y = 1
❹     if pos[3] >= self.canvas_height:
          self.y = -1
```

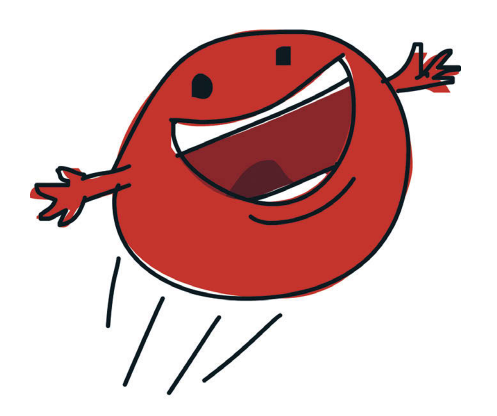
Pos — позиция, положение
Coords — координаты

В строке ❶ мы поменяли вызов функции `move`, теперь передаем в нее свойства `x` и `y`. В строке ❷ создали переменную `pos`, поместив в нее значение, полученное от функции холста `coords`. Эта функция возвращает *x*- и *y*-координаты любой фигуры на холсте по ее идентификатору. В данном случае мы передаем в функцию `coords` свойство `id`, где хранится идентификатор круга (мяча).

Функция `coords` возвращает координаты в виде списка из четырех чисел. Если вывести его на экран, мы увидим нечто подобное:

```
print(self.canvas.coords(self.id))
[255.0, 29.0, 270.0, 44.0]
```

Первые два числа (255.0 и 29.0) — это координаты левого верхнего угла прямоугольника, в который вписан наш круг (*x1* и *y1*), а вторые два (270.0 и 44.0) — координаты правого нижнего угла прямоугольника (*x2* и *y2*). Этими значениями мы и воспользуемся в следующих строках кода.



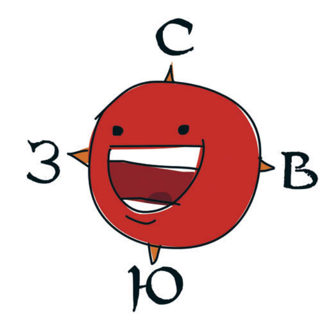


В строке ❸ сравниваем координату *y1* (это верх мяча) с нулем и, если она меньше или равна 0, задаем свойству `y` значение 1. Таким образом мы говорим, что если мяч достиг верхней границы холста, нужно прекратить двигать его вверх, перестав вычитать 1 из его вертикальной координаты. Вместо этого мы будем прибавлять к этой координате 1, чтобы мяч сменил направление.

В строке ❹ сравниваем координату *y2* (это низ мяча) со свойством `canvas_height` и, если *y2* больше или равно `canvas_height`, снова задаем свойству `y` значение –1.

Запустите этот код, и вы увидите, как мяч летает вверх-вниз, отскакивая от границ, до тех пор пока вы не закроете окно.

Меняем первоначальное направление движения мяча



Мяча, который движется то вверх, то вниз, для нашей игры недостаточно, поэтому добавим разнообразия, поменяв изначальное направление полета мяча, то есть угол его движения после запуска игры. Замените в функции `__init__` следующие строки:

```
self.x = 0
self.y = -1
```

Вместо них введите следующий код (обратите внимание на количество пробелов в начале каждой строки — их должно быть восемь):

```
❶ starts = [-3, -2, -1, 1, 2, 3]
❷ random.shuffle(starts)
❸ self.x = starts[0]
❹ self.y = -3
```

В строке ❶ мы создали переменную `starts`, поместив туда список из шести чисел. В строке ❷ перемешали элементы списка с помощью `random.shuffle`. В строке ❸ поместили в свойство `x` значение первого элемента списка. Теперь в `x` может попасть любое значение из исходного списка, от –3 до 3.

Start — старт, начало

В строке ❹ помещаем в свойство `y` значение –3 (чтобы ускорить движение мяча). Затем внесем несколько правок, чтобы мяч не улетал за боковые границы холста. Добавим в конец функции `__init__` следующую строку кода, которая, получив от функции `winfo_width` ширину холста, сохраняет ее в свойстве `canvas_width`:

```
self.canvas_width = self.canvas.winfo_width()
```

Это новое свойство пригодится в функции `draw` для проверки, не достиг ли мяч правой границы холста:

```
if pos[0] <= 0:
    self.x = 3
if pos[2] >= self.canvas_width:
    self.x = -3
```

Поскольку при достижении границ мы задаем свойству `x` значение 3 или –3, будем использовать такие же значения и для `y`, чтобы мяч всегда двигался с одинаковой скоростью. В результате функция `draw` примет такой вид:

```
def draw(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.canvas_height:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.canvas_width:
        self.x = -3
```

Сохраним код программы и запустим ее. Мяч будет перемещаться по холсту, отскакивая от его границ. Программа должна выглядеть так:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.canvas_height:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3
```

```
tk = Tk()
tk.title("Игра")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0,
highlightthickness=0)
canvas.pack()
tk.update()

ball = Ball(canvas, 'red')

while 1:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

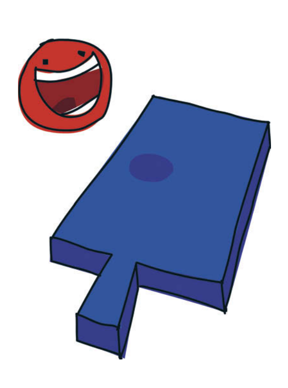
Что мы узнали

В этой главе мы начали писать первую игру, используя модуль `tkinter`. Создали класс для мяча и анимировали мяч, «научив» его летать по экрану. Проверяя координаты, добились того, чтобы мяч отскакивал от границ холста. Воспользовались функцией `shuffle` из модуля `random`, чтобы в начале игры мяч мог двигаться в разных направлениях. В следующей главе мы закончим создание игры, добавив в нее ракетку.

14

ДОДЕЛЫВАЕМ ПЕРВУЮ ИГРУ: «ПРЫГ-СКОК!»

В предыдущей главе мы начали писать первую игру: создали холст и запрограммировали движение мяча. Однако мяч бесконечно (или до тех пор, пока вы не закроете окно) летает туда-сюда по экрану. Для игры этого явно недостаточно. Добавим в программу ракетку, которой будет управлять игрок, а также возможность проигрыша. Благодаря этому игра станет сложнее и интереснее.



Создаем ракетку

Что за радость от мячика, если его нечем отбивать? Настало время создать ракетку!

Paddle — ракетка

Начнем с класса для ракетки (Paddle), код которого нужно ввести сразу после класса Ball и его функции draw:

```
def draw(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
    self.y = 3
    if pos[3] >= self.canvas_height:
        self.y = -3
    if pos[0] <= 0:
```

```
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3

class Paddle:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10,
        fill=color)
        self.canvas.move(self.id, 200, 300)

    def draw(self):
        pass
```

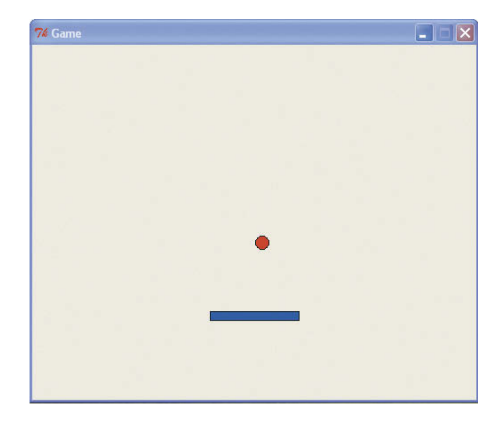
Этот код очень похож на код класса Ball. Однако вместо функции create_oval мы используем create_rectangle и перемещаем нарисованный этой функцией прямоугольник в позицию 200, 300 (200 пикселей от левого края холста и 300 пикселей от верхнего края).

Ближе к концу программы создадим объект класса Paddle и дополним игровой цикл вызовом функции draw этого объекта. Вот так:

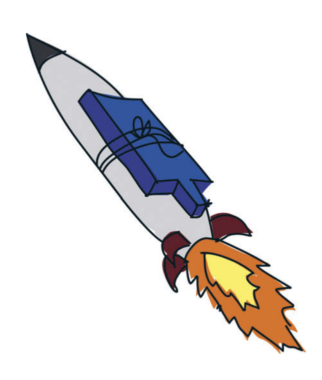
```
paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, 'red')

while 1:
    ball.draw()
    paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Если теперь запустить игру, мы увидим летающий мяч и стоящую на одном месте прямоугольную ракетку:



Управление ракеткой



Чтобы управлять ракеткой, двигая ее вправо и влево, воспользуемся привязкой к событиям: привяжем клавиши-стрелки «вправо» и «влево» к вызовам функций класса Paddle. При нажатии стрелки «влево» будем задавать свойству x объекта-ракетки значение –2 (для передвижения влево), а при нажатии стрелки «вправо» — значение 2 (для передвижения вправо).

В функции __init__ класса Paddle создадим свойства x и canvas_width для хранения ширины холста (так же, как мы это делали для класса Ball):

```
def __init__(self, canvas, color):
    self.canvas = canvas
    self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
    self.canvas.move(self.id, 200, 300)
    self.x = 0
    self.canvas_width = self.canvas.winfo_width()
```

Turn — повернуть

Добавим в класс Paddle две функции: для смены направления влево (turn_left) и вправо (turn_right). Поместим их код сразу после функции draw:

```
def turn_left(self, evt):
    self.x = -2

def turn_right(self, evt):
    self.x = 2
```

Для привязки этих функций к нажатиям нужных нам клавиш добавим в функцию __init__ класса Paddle еще две строки кода. Мы уже занимались привязкой к нажатиям клавиш в разделе «Реакция объектов на события» на стр. 183. В данном случае нужно привязать функцию turn_left к нажатию клавиши-стрелки «влево» (это событие с именем '<KeyPress-Left>'), а функцию turn_right — к клавише-стрелке «вправо» (это событие '<KeyPress-Right>'). После доработки функция __init__ будет выглядеть так:

```
def __init__(self, canvas, color):
    self.canvas = canvas
    self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
```

```
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
        self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        self.canvas.bind_all('<KeyPress-Right>', self.turn_right)
```

Код функции draw класса Paddle будет примерно таким же, как для класса Ball:

```
    def draw(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0:
            self.x = 0
        elif pos[2] >= self.canvas_width:
            self.x = 0
```

Для перемещения ракетки в соответствии со значением свойства x используем функцию move — self.canvas.move(self.id, self.x, 0). Получаем координаты ракетки (сохраняя их в переменной pos), чтобы проверить, достигла ли ракетка левой либо правой границы холста.

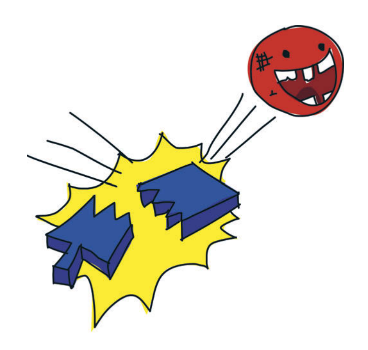
В отличие от мяча ракетка при столкновении с границей должна не отскочить от нее, а остановиться. Поэтому, если левая *x*-координата (pos[0]) меньше или равна 0 (<= 0), обнуляем свойство x (self.x = 0). Если правая *x*-координата ракетки (pos[2]) больше или равна ширине холста (>= self.canvas_width), также обнуляем x.

! *Если вы запустите игру сейчас, может понадобиться кликнуть по ее окну, чтобы программа начала реагировать на нажатия клавиш. Клик делает окно активным, и оно может обрабатывать события клавиатуры.*

Проверка на столкновение мяча с ракеткой

Сейчас наш код устроен так, что мяч не сталкивается с ракеткой, а пролетает сквозь нее. Чтобы этого не происходило, мяч должен «знать» о столкновении с ракеткой так же, как он «знает» о столкновениях с границами холста.

Эту проблему можно решить, добавив соответствующий код в функцию draw (где выполняется проверка столкновений с границами). Однако лучше создать для этого отдельные функции, чтобы программа состояла из небольших частей. Дело в том, что в программе очень сложно разобраться, если в одном месте (например,



в одной функции) скапливается слишком много строк кода. Внесем необходимые изменения.

Добавим в функцию __init__ класса Ball еще один аргумент — объект-ракетку:

```
   class Ball:
❶ def __init__(self, canvas, paddle, color):
       self.canvas = canvas
❷      self.paddle = paddle
       self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
       self.canvas.move(self.id, 245, 100)
       starts = [-3, -2, -1, 1, 2, 3]
       random.shuffle(starts)
       self.x = starts[0]
       self.y = -3
       self.canvas_height = self.canvas.winfo_height()
       self.canvas_width = self.canvas.winfo_width()
```

Обратите внимание, что в строке ❶ мы изменили аргументы __init__, добавив к ним ракетку (paddle), а в строке ❷ сохранили значение аргумента paddle в свойстве с таким же названием.

Теперь нужно изменить код создания объекта-мяча с учетом нового аргумента — ракетки. Этот код находится в конце программы перед главным циклом:

```
paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')

while 1:
    ball.draw()
    paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Hit paddle — *здесь* столкновение с ракеткой

Код для проверки столкновения с ракеткой будет сложнее, чем код проверки для границ холста. Поместим его в новую функцию hit_paddle, добавив ее вызов в функцию draw класса Ball, рядом с проверкой на столкновение с нижней границей:

```
def draw(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.canvas_height:
```

```
            self.y = -3
        if self.hit_paddle(pos) == True:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
            if pos[2] >= self.canvas_width:
            self.x = -3
```

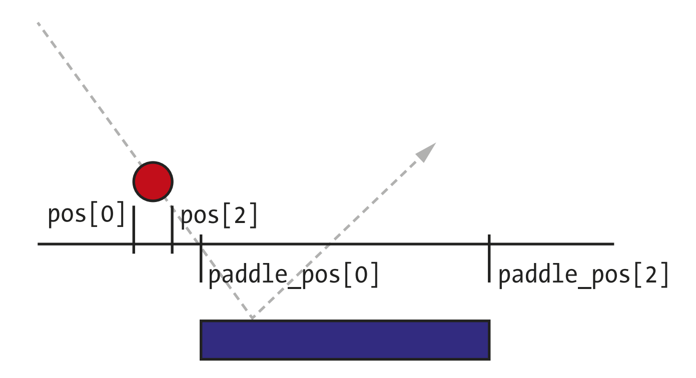
Если hit_paddle возвращает True, мы меняем направление полета мяча, задавая свойству y значение –3 (self.y = -3). Не пытайтесь сейчас запустить игру — мы еще не создали функцию hit_paddle. Давайте это исправим.

Добавьте код функции hit_paddle сразу перед функцией draw:

```
❶ def hit_paddle(self, pos):
❷     paddle_pos = self.canvas.coords(self.paddle.id)
❸     if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
❹         if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
              return True
      return False
```

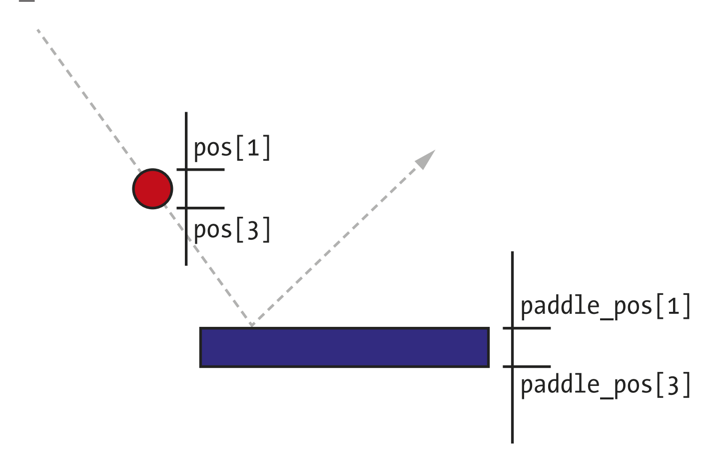
В строке ❶ объявляем функцию с аргументом pos, в котором будем передавать текущие координаты мяча. В строке ❷ получаем координаты ракетки и сохраняем их в переменной paddle_pos.

В строке ❸ находится конструкция if, условие которой означает: *x*-координата правой стороны мяча больше, чем *x*-координата левой стороны ракетки, и *x*-координата левой стороны мяча меньше, чем *x*-координата правой стороны ракетки? Здесь pos[2] соответствует *x*-координате правой стороны мяча, а pos[0] — *x*-координате его левой стороны. При этом paddle_pos[0] соответствует *x*-координате левой стороны ракетки, а paddle_pos[2] — *x*-координате ее правой стороны. Вот схема, где показаны эти координаты для случая, когда мяч вот-вот коснется ракетки.



Мяч летит к ракетке, но его правая сторона (pos[2]) еще не достигла левой стороны ракетки (paddle_pos[0]).

В строке ❹ проверяем, не находится ли нижняя сторона мяча (pos[3]) между верхом ракетки (paddle_pos[1]) и ее низом (paddle_pos[3]). На следующей схеме показана ситуация, когда нижняя сторона мяча (pos[3]) еще не достигла верхней стороны ракетки (paddle_pos[1]).



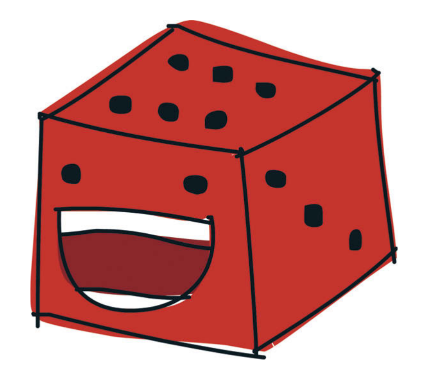
Для этого случая функция hit_paddle вернет значение False.

> **!** *Зачем проверять, не находится ли низ мяча между верхом и низом ракетки, если можно просто проверить низ мяча на столкновение с верхом ракетки? Дело в том, что мы перемещаем мяч шагами по 3 пикселя, и низ мяча может «перескочить» за верх ракетки. Если в этой ситуации сравнить только низ мяча и верх ракетки, проверка даст отрицательный результат и мяч полетит дальше сквозь ракетку.*

Добавим возможность проигрыша

Пора сделать из программы с летающим мячом и ракеткой настоящую игру. Для игр важен элемент неопределенности, то есть вероятность проигрыша, а сейчас мяч просто летает по экрану, и проиграть невозможно.

Закончим создание игры, написав код, останавливающий анимацию, если мяч коснется «земли» (то есть нижней границы холста).



Сперва создадим в теле функции `__init__` класса `Ball` свойство `hit_bottom` (признак того, что мяч достиг нижней границы холста). Добавим этот код в самый низ функции `__init__`:

Hit bottom — удар о дно

```
self.canvas_height = self.canvas.winfo_height()
self.canvas_width = self.canvas.winfo_width()
self.hit_bottom = False
```

Изменим главный цикл в конце программы следующим образом:

```
while 1:
    if ball.hit_bottom == False:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Теперь на каждом повторе цикла проверяем значение `hit_bottom`, чтобы узнать, не достиг ли мяч нижней границы холста. Как видно из условия `if`, мяч и ракетка будут перемещаться, только если мяч не достиг нижней границы. В противном случае мяч и ракетка замрут (мы больше не будем их анимировать), что и будет означать конец игры.

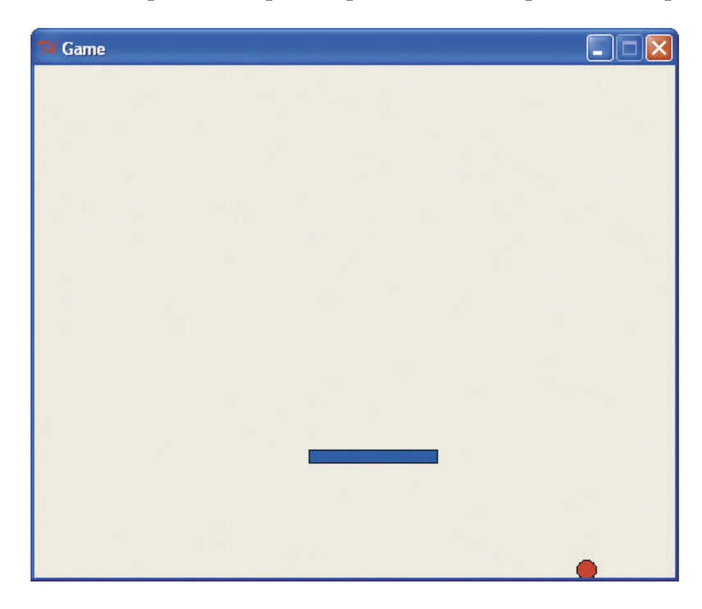
Осталось лишь доработать функцию `draw` класса `Ball`:

```
def draw(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
    self.y = 3
    if pos[3] >= self.canvas_height:
        self.hit_bottom = True
    if self.hit_paddle(pos) == True:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.canvas_width:
        self.x = -3
```

Мы изменили конструкцию `if`, проверяющую, не достиг ли мяч нижней границы холста: не сравнялась ли *y*-координата низа мяча с высотой холста (`canvas_height`) или не превысила ли она это значение. Если это произошло, вместо изменения свойства `y` (соприкосновение с нижней

границей холста больше не должно приводить к отскоку мяча) задаем свойству `hit_bottom` значение `True`.

Если теперь запустить игру и позволить мячу пролететь мимо ракетки, все движение на экране должно остановиться, то есть соприкосновение мяча с низом игрового экрана приведет к завершению игры.



В итоге код игры должен выглядеть следующим образом (если игра не работает или работает неправильно, сверьте с ним свою программу):

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, paddle, color):
        self.canvas = canvas
        self.paddle = paddle
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False

    def hit_paddle(self, pos):
```

```
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
            if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
                return True
        return False

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if self.hit_paddle(pos) == True:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3

class Paddle:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
        self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        self.canvas.bind_all('<KeyPress-Right>', self.turn_right)

    def draw(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0:
            self.x = 0
        elif pos[2] >= self.canvas_width:
            self.x = 0

    def turn_left(self, evt):
        self.x = -2

    def turn_right(self, evt):
        self.x = 2

tk = Tk()
tk.title("Игра")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0,
highlightthickness=0)
canvas.pack()
```

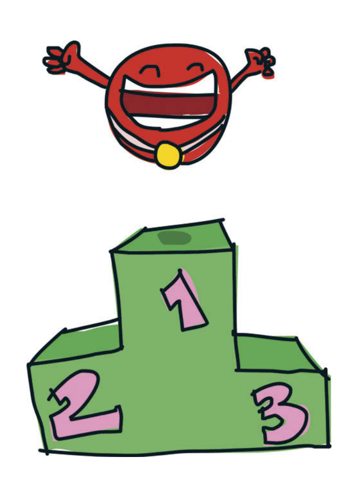
```
tk.update()

paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')

while 1:
    if ball.hit_bottom == False:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Что мы узнали

В этой главе мы закончили писать первую игру, созданную с помощью модуля `tkinter`. Создали класс для ракетки, научились проверять столкновение мяча с ракеткой и границами игрового холста. Использовали привязку к событиям, чтобы управлять ракеткой нажатиями на клавиши-стрелки «влево» и «вправо», и анимировали ракетку, добавив вызов ее функции `draw` в главный цикл. Добавили возможность проигрыша, так чтобы при столкновении мяча с нижней границей холста (когда игроку не удалось отбить мяч ракеткой) игра завершалась.



Упражнения

Сейчас игра крайне проста. Чтобы она выглядела более профессионально, стоит ее доработать.

#1. Задержка перед началом игры

Игра начинается, едва запустившись, однако без клика по холсту клавиши-стрелки могут не распознаваться программой. Попробуйте добавить задержку перед стартом игры, чтобы у игрока хватило времени кликнуть по холсту. А еще лучше, воспользовавшись привязкой к событиям, сделать так, чтобы игра начиналась при клике мышкой внутри окна.

Подсказка 1: мы уже добавляли код для привязки к событиям в класс ракетки (`Paddle`), его можно использовать как отправную точку.

Подсказка 2: для привязки к нажатию левой кнопки мышки используйте имя события `'<Button-1>'`.

Button — кнопка

#2. Экран «Конец игры»

Сейчас по окончании игры экран застывает. Пользователю будет гораздо удобнее, если при касании мячиком нижнего края холста на экране появится сообщение «Конец игры». В этом вам поможет функция `create_text`. Стоит обратить внимание и на функцию `itemconfig` с аргументом `state`, в котором можно передавать такие значения, как `normal` и `hidden` (см. раздел «Для чего еще нужен идентификатор» на стр. 186). В качестве дополнительного задания попробуйте выдержать перед показом сообщения небольшую паузу.

Item config — конфигурация элемента
State — состояние
Normal — обычный
Hidden — скрытый

#3. Ускорение мяча

Если вам доводилось играть в теннис, вы знаете, что порой мяч отлетает от ракетки быстрее, чем двигался раньше. Это зависит от силы удара по нему. В нашей игре мяч движется с постоянной скоростью. Доработайте программу так, чтобы скорость мяча при отскоке менялась в зависимости от движений ракетки.

#4. Счет в игре

А не добавить ли в игру подсчет очков? Каждый раз, когда мяч отскакивает от ракетки, счет должен расти. Сделайте так, чтобы набранные очки отображались в правом верхнем углу игрового экрана. Здесь вам пригодится функция `itemconfig` (см. раздел «Для чего еще нужен идентификатор» на стр. 186).



ЧАСТЬ III

Пишем игру «Человечек спешит к выходу»

15

СОЗДАЕМ ГРАФИКУ ДЛЯ ИГРЫ ПРО ЧЕЛОВЕЧКА

Перед созданием игры (или другой программы) полезно наметить план действий. Он должен содержать описание игры, ее основных элементов и персонажей. Когда дело дойдет до программирования, план поможет сосредоточиться на цели разработки. А результат может и не совпадать с изначальным планом. Это нормально.

Приступим к созданию забавной игры «Человечек спешит к выходу».

План игры про человечка

Вот описание нашей новой игры:

- Человечек очутился в плену у злодея, и ваша задача — помочь человечку спастись, добравшись до выхода на верхнем этаже.
- Человечек выглядит как фигурка, которая может двигаться вправо, влево, а также прыгать. На этажах расположены платформы, на которые человечку предстоит запрыгивать.
- Цель игры — добраться до двери выхода.



Из этого описания ясно, что нам понадобится несколько изображений: для человечка, платформ и двери. Также необходим код игры, однако сначала займемся созданием графики, чему и посвящена эта глава.

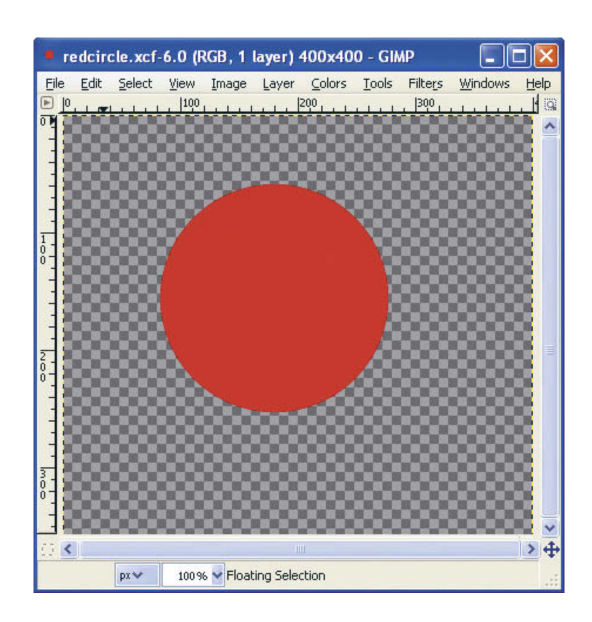
Как мы будем создавать игровые изображения? Можно воспользоваться графическими примитивами, как мы делали в предыдущих главах, рисуя мяч и ракетку. Но для этой игры нужна графика посложнее, поэтому будем использовать спрайты.

Спрайты — это графические объекты в игре, чаще всего персонажи. Как правило, спрайты создаются заранее (то есть это готовые на момент запуска игры изображения), а не рисуются в программном коде, как было в игре с мячом и ракеткой. Человечек и платформы будут спрайтами. А чтобы создать для них изображения, нужно установить графический редактор.

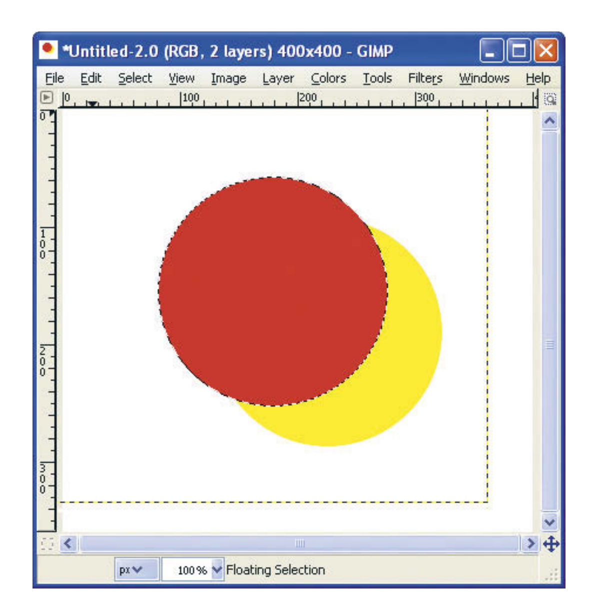
Устанавливаем GIMP

Существует немало графических редакторов, но для этой игры понадобится редактор с поддержкой *прозрачности* (или, как говорят, *альфа-канала*), что позволит создавать изображения с областями, которые не отображаются на экране.

Прозрачные области нужны, чтобы при движении спрайта по экрану его фон не затирал спрайты на заднем плане. На этом рисунке шахматной клеткой показана прозрачная фоновая область:



Если скопировать данное изображение и вывести поверх другого, фон не будет ничего заслонять:



GIMP (*http://www.gimp.org/*) — это бесплатный графический редактор для систем Ubuntu, Mac OS X и Windows, который поддерживает прозрачность. Скачайте и установите его.

- Если вы используете Windows, программу установки можно скачать на странице http://www.gimp.org/downloads/.
- Чтобы установить GIMP в Ubuntu, откройте центр приложений Ubuntu и введите в строке поиска «gimp». В списке результатов найдите приложение «GIMP Image Editor» и кликните **Install**.
- Для Mac OS X скачайте установочный пакет со страницы *http://gimp.lisanet.de/Website/Download.html*.

Также для нашей игры нужно создать папку: наведите мышку на пустое место рабочего стола, нажмите правую кнопку и выберите **New ▸ Folder** (**Создать ▸ Папку**). В Ubuntu этот пункт меню называется **Create New Folder**, а в MAC OS X — **New Folder**. В окне выбора имени файла введите *stickman*.

Stickman — человечек, нарисованный линиями

Создаем изображения для игры

Мы установили графический редактор — пора рисовать. Создадим изображения для следующих игровых элементов:

- человечка, который может двигаться вправо, влево и прыгать;

- платформ трех разных размеров;
- двери — открытой и закрытой;
- фона (если мы хотим, чтобы игра выглядела красиво, сплошной белый или серый фон нам не подходит).

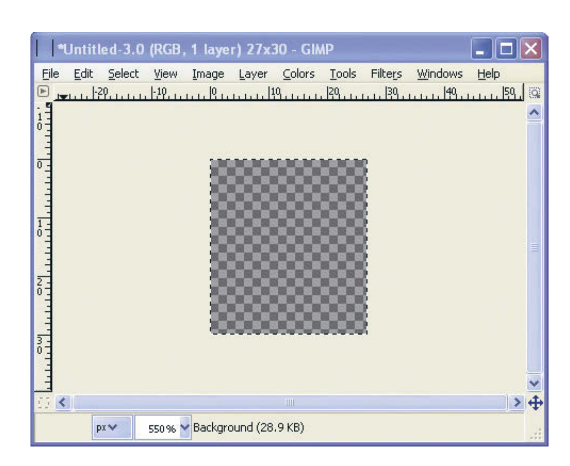
Для начала разберемся, как подготовить для изображений прозрачный фон.

Подготовка прозрачного фона

Чтобы создать изображение с прозрачностью (с альфа-каналом), запустите GIMP и выполните следующие действия:

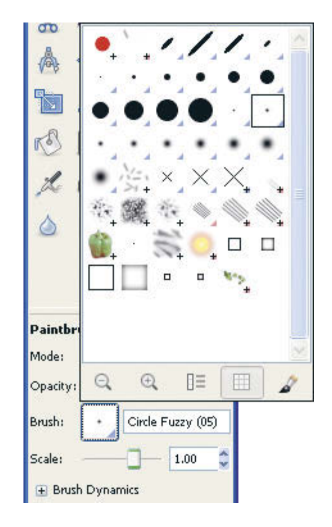
1. Выберите в меню **File ▸ New** (**Файл ▸ Создать**).
2. В диалоговом окне выберите ширину изображения 27 пикселей и высоту 30 пикселей и кликните **OK**.
3. Выберите **Layer ▸ Transparency ▸ Add Alpha Channel** (**Слой ▸ Прозрачность ▸ Добавить альфа-канал**).
4. Выберите **Select ▸ All** (**Выделение ▸ Выделить все**).
5. Выберите **Edit ▸ Cut** (**Правка ▸ Вырезать**).

Должно получиться прозрачное изображение. Чередующиеся светло- и темно-серые квадратики, как на этом скриншоте, обозначают прозрачность (показано в увеличении).



Теперь можно рисовать человечка.

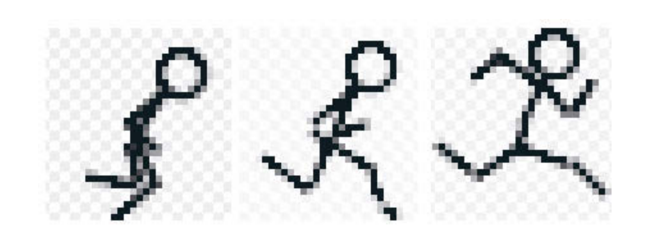
Рисуем человечка



Кликните по инструменту «Кисть» в палитре инструментов GIMP и выберите кисть в виде маленькой точки (список кистей обычно находится в правом нижнем углу экрана), как показано на скриншоте справа.

Для движения вправо создадим три отдельных изображения (кадра) и будем использовать их для анимирования бегущего и прыгающего человечка (вспомните, как мы создавали анимацию в главе 12).

В увеличении эти изображения должны выглядеть примерно так:



Рисунки могут быть другими, главное, чтобы они соответствовали трем фазам движения фигурки. Размер каждого кадра — 27 пикселей в ширину и 30 в высоту.

Человечек бежит вправо

Подготовим кадры для бега вправо. Создайте первое изображение:

1. Нарисуйте первую фазу движения фигурки (крайнее слева изображение на предыдущем рисунке).
2. Выберите **File ▸ Export As** (**Файл ▸ Экспортировать как**).
3. В диалоговом окне введите имя файла *figure-R1.gif*, кликните значок плюс (+) с подписью **Select File Type** (**Выберите тип файла**).
4. В появившемся списке выберите **GIF image** (**Изображение GIF**).
5. Сохраните файл в созданную ранее папку *stickman* (отыщите ее в списке папок).

Figure — фигура

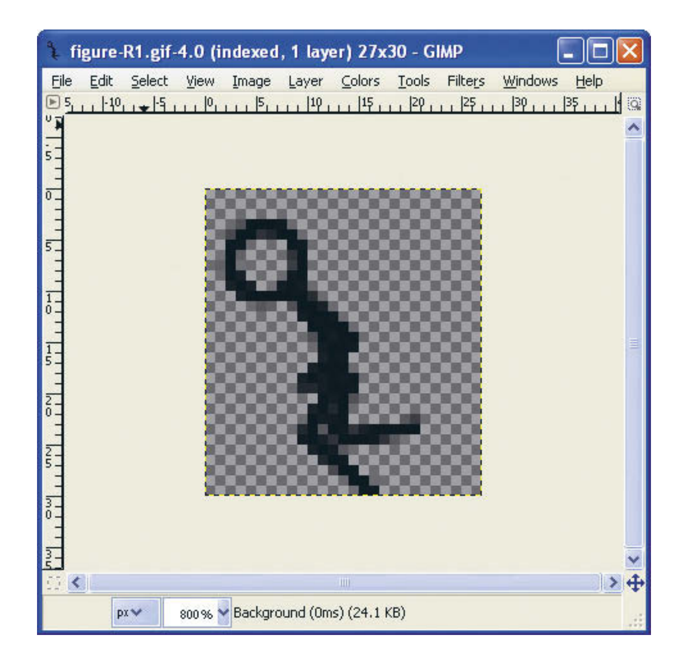


Таким же образом создайте еще одно изображение размером 27 на 30 пикселей и нарисуйте следующую фазу движения человечка. Сохраните файл под именем *figure-R2.gif*. Затем повторите эти действия, нарисовав последнюю фазу движения вправо, и сохраните файл под именем *figure-R3.gif*.

Человечек бежит влево

Вместо того чтобы рисовать фазы движения влево заново, можно зеркально перевернуть фазы движения вправо с помощью GIMP.

По очереди откройте в GIMP каждое из предыдущих изображений и выберите **Tools ▸ Transform Tools ▸ Flip** (**Инструменты ▸ Преобразование ▸ Зеркало**). Теперь при клике по изображению оно должно меняться на зеркальное. Сохраните перевернутые фазы движения фигурки под именами *figure-L1.gif*, *figure-L2.gif* и *figure-L3.gif*.



Мы создали шесть изображений человечка. Теперь нарисуем платформы и дверь.

Рисуем платформы

Создадим три платформы высотой по 10 пикселей и разной ширины: 100 пикселей, 66 пикселей и 32 пикселя. Рисуйте изображения, как вам нравится, но их фон должен быть прозрачным, как и у бегущих фигурок.

Платформы могут выглядеть так (в увеличении):

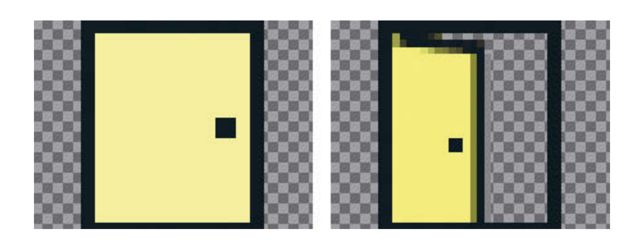


Сохраните изображения в той же папке *stickman*, где находятся изображения фигурки. Самую длинную платформу назовите *platform1.gif*, среднюю — *platform2.gif*, а самую маленькую — *platform3.gif*.

Platform — платформа

Рисуем дверь

Размер двери должен соответствовать размеру человечка (27 пикселей в ширину и 30 в высоту), причем нам нужна пара изображений: одно для закрытой двери, второе — для открытой. Выглядеть двери могут примерно так (в увеличении):



Создайте эти изображения следующим образом:



1. Кликните внутри прямоугольника с цветом фона (он находится в нижней части палитры инструментов GIMP) — появится диалог выбора цвета. Подберите для двери цвет на свой вкус. На иллюстрации справа выбран желтый.

2. Выберите инструмент «Ведро» (обведен рамкой на иллюстрации) и заполните изображение выбранным цветом.

3. Выберите черный цвет фона.

4. Выберите инструмент «Карандаш» либо «Кисть» (справа от инструмента «Ведро») и нарисуйте черный контур двери, а также дверную ручку.

Door — дверь

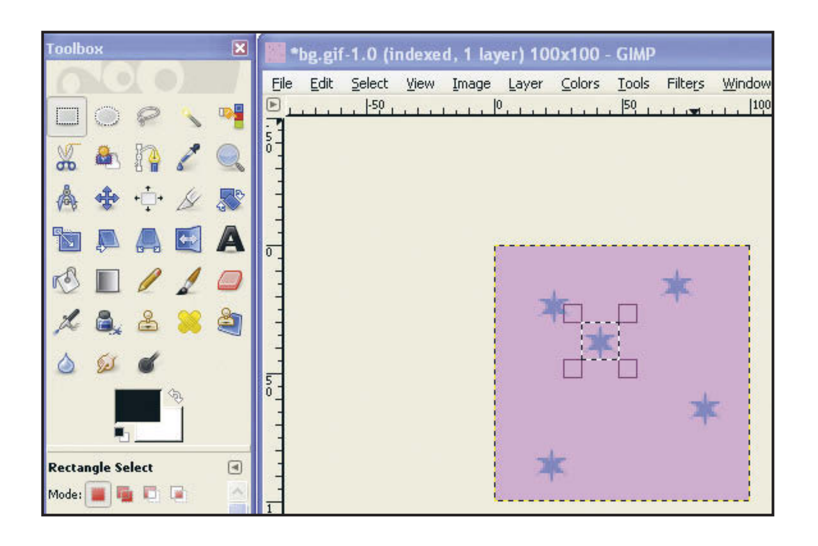
5. Сохраните изображения закрытой и открытой двери в папку *stickman* под именами *door1.gif* и *door2.gif*.

Рисуем фон

И наконец, нужно создать изображение для фона. Пусть оно будет размером 100 на 100 пикселей. На этот раз прозрачность нам не нужна. Заполним фон цветом, чтобы получились «обои», которые будут отображаться позади остальных игровых элементов.

Чтобы нарисовать фоновые обои, выберите **File ▸ New** (**Файл ▸ Создать**) и задайте размер изображения: 100 пикселей в ширину и 100 в высоту. Подберите цвет фона (не забывайте, это обои в логове Злодея). Я выбрал темно-розовый.

Можете украсить обои на свой вкус — цветочками, полосками, звездочками и так далее. Если вы хотите сделать обои со звездами, выберите другой цвет, кликните по инструменту «Карандаш» и нарисуйте первую звездочку. Затем, кликнув по инструменту «Прямоугольное выделение», выделите часть изображения вокруг звезды, скопируйте и несколько раз вставьте в разных местах картинки. Для этого выберите **Edit ▸ Copy** (**Правка ▸ Копировать**), а затем — несколько раз **Edit ▸ Paste** (**Правка ▸ Вставить**). После вставки фрагмент изображения можно двигать, кликнув по нему мышкой. Вот обои, разукрашенные синими звездочками (в палитре выбран инструмент «Прямоугольное выделение»).

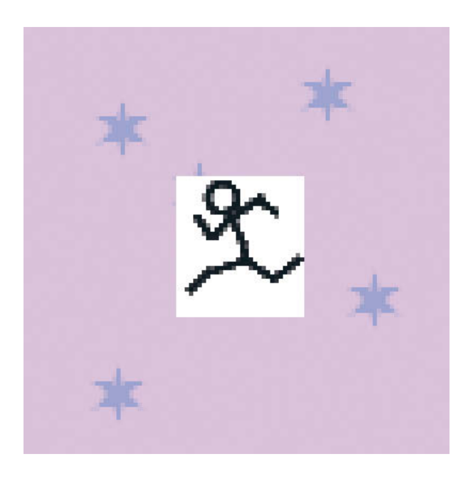


Background — задний план

Закончив рисовать, сохраните изображение в папку *stickman* под именем *background.gif*.

Прозрачность

Теперь, когда игровая графика готова, можно убедиться, насколько важна прозрачность для всех изображений (за исключением обоев). Посмотрим, что будет, если поместить человечка на передний план, перед обоями, не сделав его фон прозрачным:



Белый фон изображения человечка перекрыл собой часть обоев. Однако если использовать для фигурки прозрачный фон, результат будет другим:

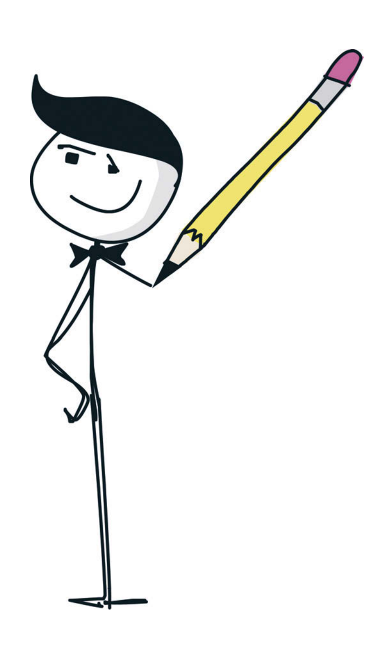


Фигурка накладывается на обои, ничего не заслоняя. Согласитесь, так игра будет выглядеть гораздо лучше!

Что мы узнали

В этой главе мы написали краткий план игры «Человечек спешит к выходу» и выяснили, с чего стоит начинать ее разработку. Первым делом для игры нужны графические изображения, поэтому мы создали необходимую графику с помощью графического редактора. Научились делать фон изображений прозрачным, чтобы он не заслонял другие изображения на игровом экране.

В следующей главе создадим несколько классов для игры.



16

РАЗРАБОТКА ИГРЫ

Создав все необходимые изображения для игры «Человечек спешит к выходу», можем приступать к написанию кода. План игры из предыдущей главы подсказывает, что нужно запрограммировать: фигурку бегающего и прыгающего человечка, а также платформы, чтобы запрыгивать на них.

Прежде чем писать код для отрисовки человечка и платформ, создадим холст с обоями в качестве фона.

Создаем класс игры

Начнем с главного класса игры (назовем его Game), который будет управлять другим кодом. В классе Game будет функция __init__ для инициализации игры, а также главный цикл игровой анимации.

Game — игра

Настраиваем заголовок окна и создаем холст

В начале функции __init__ зададим текст заголовка окна и создадим холст. Это напоминает аналогичный код для игры «Прыг-скок!» из главы 13. Откройте окно редактора, введите следующий код и сохраните файл под именем *stickmangame.py* (он должен находиться в папке *stickman*, которую мы создали в главе 15).

```
from tkinter import *
import random
import time
class Game:
```

```
    def __init__(self):
        self.tk = Tk()
        self.tk.title("Человечек спешит к выходу")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                    highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
```

В первой части кода (от `tkinter import *` до `self.tk.wm_attributes`) создаем объект `tk` и с помощью `self.tk.title` устанавливаем заголовок окна («Человечек спешит к выходу»). Вызываем функцию `resizable`, чтобы задать фиксированный размер окну, и размещаем его поверх остальных окон с помощью функции `wm_attributes`.

Создаем холст (`self.canvas = Canvas`) и вызываем функции `pack` и `update` объекта `tk`. Кроме того, для класса `Game` нужны два свойства — `height`, где будет храниться высота холста, и `width`, куда сохраним его ширину.

! *Обратный слеш (\) в строке* `self.canvas = Canvas` *выполняет роль знака переноса. Таким образом можно расположить длинную строку кода на нескольких экранных строках, если код не помещается по ширине страницы.*

Дописываем функцию __init__

Добавим в файл *stickmangame.py* оставшийся код функции `__init__`. Он служит для загрузки и отображения на холсте фонового рисунка (обоев).

```
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
❶       self.bg = PhotoImage(file="background.gif")
❷       w = self.bg.width()
        h = self.bg.height()
❸       for x in range(0, 5):
❹           for y in range(0, 5):
❺               self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
❻       self.sprites = []
        self.running = True
```

В строке ❶ мы создали свойство `bg`, которое содержит объект `PhotoImage` (это фоновый рисунок *background.gif*, который мы создали в главе 15). Начиная со строки ❷, сохраняем размеры изображения в переменных `w` и `h`. Функции `width` и `height` класса `PhotoImage` возвращают ширину и высоту изображения соответственно (на момент их вызова изображение должно быть загружено).

Далее следуют два вложенных друг в друга цикла. Чтобы понять, как они работают, представьте, что у вас есть резиновая печать, подушечка с чернилами и большой лист бумаги. Как заполнить лист цветными отпечатками? Можно беспорядочно ставить отпечатки, пока лист не заполнится. Это займет немало времени, а результат будет выглядеть очень неаккуратно. Однако есть другой вариант: ставить отпечатки бок к боку, от верхнего края листа к нижнему и, заполнив колонку, переходить к следующей, возвращаясь к верхнему краю листа, как показано на рисунке справа.

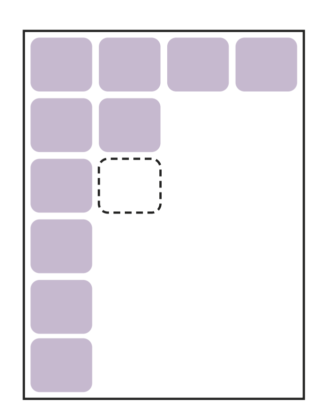


Рисунок обоев, который мы создали в предыдущей главе, — это печать. Размер холста — 500 на 500 пикселей, а фоновое изображение — квадрат со стороной 100 пикселей. Значит, чтобы заполнить игровой экран «отпечатками» изображения, нужно расположить их в пять строк и в пять столбцов. Цикл в строке ❸ перебирает столбцы (слева направо), а цикл ❹ — строки (сверху вниз).

В строке ❺ умножаем переменную первого цикла (`x`) на ширину изображения (`x * w`), чтобы получить отступ от левого края холста, и умножаем переменную второго цикла (`y`) на высоту изображения (`y * h`), чтобы получить отступ от верхнего края. Для вывода изображения в найденной позиции используем функцию холста `create_image`.

В строке ❻ создаем свойство `sprites`, содержащее пустой список, и свойство `running`, где хранится булево значение `True`. Эти свойства пригодятся нам позже.

Создаем функцию mainloop

Функция `mainloop` класса `Game` (которая очень похожа на главный цикл в игре «Прыг-скок!» из главы 13) будет управлять игровой анимацией. Введите такой код:

Main loop — основной цикл

```
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h, \
                    image=self.bg, anchor='nw')
```

```
        self.sprites = []
        self.running = True

    def mainloop(self):
❶       while 1:
❷           if self.running == True:
❸               for sprite in self.sprites:
❹                   sprite.move()
❺           self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)
```

Со строки ❶ начинается цикл `while`, который будет работать до закрытия окна игры. В строке ❷ проверяем: свойство `running` равно `True`? Если да, в строке ❸ перебираем в цикле все спрайты из списка `sprites` (`self.sprites`), вызывая для каждого из них функцию `move` (сейчас этот код бесполезен, ведь мы еще не добавили в список ни одного спрайта, но позднее он нам пригодится).

Последние три строки кода, начиная со строки ❺, нужны для принудительной перерисовки экрана с помощью объекта `tk`, после чего делается пауза на сотую долю секунды — так же, как в игре «Прыг-скок!».

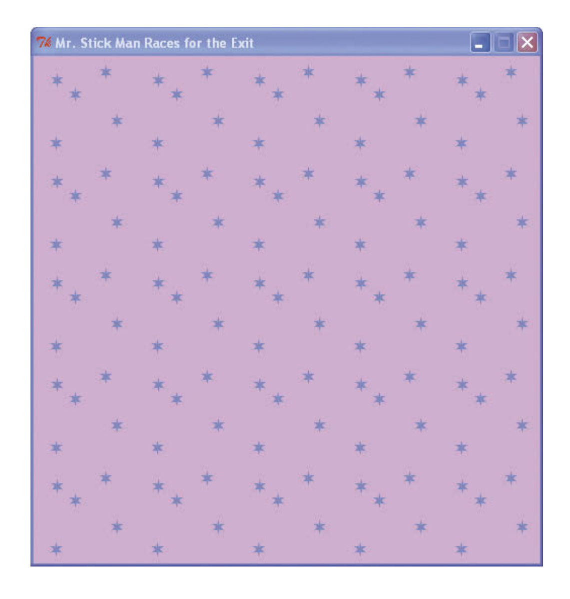
Чтобы запустить код, добавьте в программу еще две строчки (обратите внимание: их нужно вводить без отступов) и сохраните файл:

```
g = Game()
g.mainloop()
```

! *Этот код нужно добавить в самый конец файла. Убедитесь, что созданные ранее изображения находятся в той же папке, что и файл с программой. Если в главе 15 вы создали папку stickman и сохранили в нее все изображения, файл с программой на языке Python должен находиться там же.*

В этих двух строках кода создаем объект класса `Game` и сохраняем его в переменной `g`. Вызываем функцию `mainloop` созданного объекта, чтобы заработал главный цикл игры.

Сохранив программу, запустите ее из IDLE, выбрав **Run ▸ Run Module**. Должно появиться окно с холстом, который заполнен фоновым изображением.

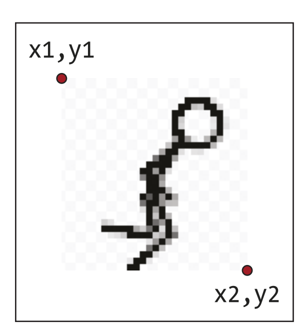


Итак, мы добавили в игру отображение фона и цикл игровой анимации, который будет выводить на экран спрайты (когда мы их создадим).

Создаем класс Coords

Создадим класс, который пригодится для размещения спрайтов на экране. В объектах этого класса будут храниться позиции всех графических элементов в игре, то есть координаты левого верхнего угла (*x1* и *y1*) и правого нижнего угла (*x2* и *y2*) для каждого из спрайтов.

Задать позицию изображения с помощью этих координат можно так:



Назовем наш новый класс `Coords`; в нем будет одна лишь функция `__init__`, принимающая четыре аргумента (`x1`, `y1`, `x2` и `y2`). Вот код, который следует добавить в начало файла *stickmangame.py*:

Coords — от coordinates — координаты

```
class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

Обратите внимание, что мы сохраняем каждый аргумент в свойстве с таким же именем (`x1`, `y1`, `x2`, и `y2`). Объекты класса `Coords` уже очень скоро нам понадобятся.

Проверка столкновений

Теперь мы знаем, как хранить позиции спрайтов в игре, но помимо этого нужно проверять спрайты на столкновение друг с другом. Например, человечек, прыгая по экрану, может столкнуться с одной из платформ. Сперва разберемся, как проверять пересечения координат по вертикали и по горизонтали, а затем используем созданные процедуры для проверки столкновения двух спрайтов.

Пересечение по горизонтали

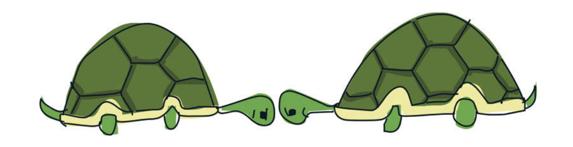
Within — в пределах

Создадим функцию `within_x`, которая определяет, пересекается ли одна пара *x*-координат (*x1* и *x2*) с другой парой *x*-координат (опять же, *x1* и *x2*). Сделать это можно несколькими способами. Вот простой вариант, подходящий для нашей игры, — добавьте этот код в программу после кода класса `Coords`:

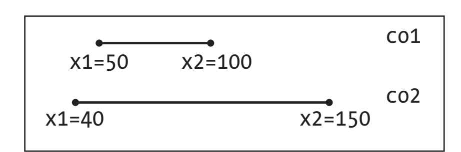
```
class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def within_x(co1, co2):
❶    if co1.x1 > co2.x1 and co1.x1 < co2.x2:
❷        return True
❸    elif co1.x2 > co2.x1 and co1.x2 < co2.x2:
❹        return True
❺    elif co2.x1 > co1.x1 and co2.x1 < co1.x2:
        return True
❻    elif co2.x2 > co1.x1 and co2.x2 < co1.x2:
        return True
❼    else:
❽        return False
```

Функция `within_x` принимает аргументы `co1` и `co2` (и тот и другой — объекты класса `Coord`). В строке ❶ проверяем, находится ли левая *x*-координата первого объекта (`co1.x1`) между левой и правой *x*-координатами второго объекта. Если это так, в строке ❷ возвращаем `True`.



Посмотрите на два отрезка, *x*-координаты которых пересекаются: *x*-координаты начала каждого из отрезков — это `x1`, а *x*-координаты конца — `x2`.



Первый отрезок на схеме (`co1`) начинается с позиции 50 (`x1`) и заканчивается в позиции 100 (`x2`). Второй отрезок (`co2`) начинается с позиции 40 и заканчивается в позиции 150. Поскольку в данном случае координата `x1` первого отрезка находится между координатами `x1` и `x2` второго отрезка, первая из конструкций `if` сработает и функция вернет `True`.

В строке ❸ проверяем, находится ли правая координата первого отрезка (`co1.x2`) между левой (`co2.x1`) и правой (`co2.x2`) координатами второго отрезка. Если это так, в строке ❹ функция возвращает значение `True`. Проверки `elif` в строках ❺ и ❻ нужны примерно для этого же. Однако теперь левая и правая координаты второго отрезка (`co2`) сравниваются с координатами первого отрезка (`co1`).

Если ни одна из этих проверок не сработает, будет выполнен вариант `else` в строке ❽ и мы вернем из функции значение `False`, что означает: горизонтальные координаты этих отрезков не пересекаются.

Чтобы посмотреть на функцию `within_x` в действии, вернемся к схеме, где изображены два отрезка. Координаты `x1` и `x2` первого из них — 40 и 100, а `x1` и `x2` второго — 50 и 150. Вызовем функцию `within_x` для этого набора координат:

```
>>> c1 = Coords(40, 40, 100, 100)
>>> c2 = Coords(50, 50, 150, 150)
>>> print(within_x(c1, c2))
True
```

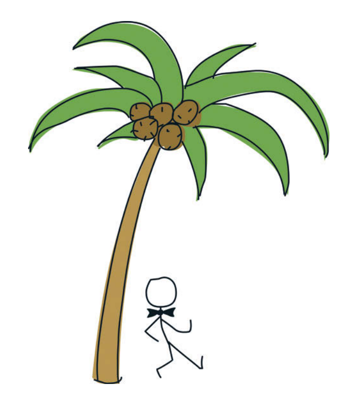
Функция вернула True. Это первый шаг к проверке столкновения двух спрайтов. Теперь, создав классы для фигурки человечка и платформ, мы сможем узнать, пересекаются ли их *x*-координаты.

Несколько конструкций elif, которые возвращают одно и то же значение, — не очень хороший стиль программирования. Можно сократить код функции within_x, заключив каждое из условий в скобки и объединив их с помощью ключевого слова or. Если хотите, чтобы функция стала короче и аккуратнее, замените ее код таким:

```
def within_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x2):
        return True
    else:
        return False
```

Мы снова воспользовались символом «обратный слеш» (\), чтобы разбить очень длинную строку с условиями на несколько частей.

Пересечение по вертикали



При определении столкновений нужно учитывать и вертикальные координаты. Функция within_y очень похожа на функцию within_x — мы проверяем, находится ли координата y1 первого спрайта между координатами y1 и y2 второго и так далее. Добавьте код этой функции в программу следом за функцией within_x. Напишем короткий вариант с объединением условий по or (или):

```
def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y2):
        return True
    else:
        return False
```

Собираем по частям код проверки столкновений

Мы можем определить, пересекается ли одна пара *x*-координат с другой, и то же самое — для *y*-координат. Теперь можно написать код, проверяющий, столкнулся ли один спрайт с другим и какой стороной. Создадим для этого функции `collided_left`, `collided_right`, `collided_top` и `collided_bottom`.

Collide — столкновение

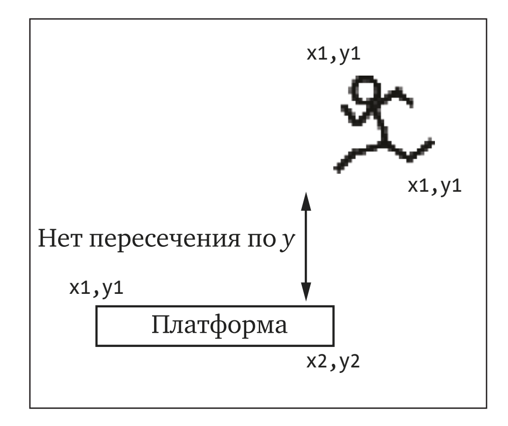
Функция collided_left

Добавьте код функции `collided_left` (столкновение слева) в программу после функций `within_x` и `within_y`:

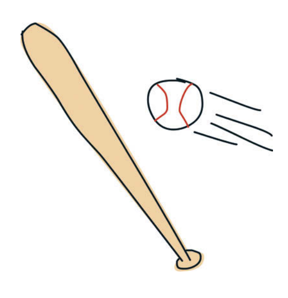
```
❶ def collided_left(co1, co2):
❷     if within_y(co1, co2):
❸        if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
❹            return True
❺     return False
```

Эта функция возвращает `True`, если один объект класса `Coords` столкнулся левой стороной (координата `x1`) с другим объектом.

Функция принимает два аргумента: `co1` (первый координатный объект) и `co2` (второй координатный объект). В строке ❷ проверяем, пересекаются ли наши объекты по вертикали. Вызываем для этого функцию `within_y`. Если человечек находится над платформой, нет смысла проверять, столкнулся он с ней или нет (см. рисунок):



В строке ❸ проверяем, находится ли левая горизонтальная координата первого координатного объекта (`co1.x1`) между правой и левой координатами второго объекта (`co2.x2` и `co2.x1`), что соответствует столкновению левой стороны первого объекта со вторым. Если это так,



в строке ❹ функция возвращает значение True. В противном случае в строке ❺ возвращаем False.

Функция collided_right

Функция collided_right определяет столкновение правой стороны первого объекта со вторым и очень похожа на collided_left:

```
   def collided_right(co1, co2):
❶      if within_y(co1, co2):
❷          if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
❸              return True
❹      return False
```

Так же, как и в коде collided_left, в строке ❶ проверяем пересечение объектов по вертикали с помощью функции within_y. В строке ❷ проверяем, находится ли значение x2 первого объекта в промежутке между координатами x1 и x2 второго объекта, возвращая в строке ❸ True, если это так. Если нет, возвращаем False в строке ❹.

Функция collided_top

Функция collided_top, проверяющая столкновение верхней стороны первого объекта со вторым, устроена аналогично двум предыдущим функциям.

```
   def collided_top(co1, co2):
❶      if within_x(co1, co2):
❷          if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
               return True
       return False
```

Однако на этот раз мы первым делом проверяем объекты на пересечение по горизонтали, вызывая в строке ❶ функцию within_x. Затем в строке ❷ проверяем, находится ли верхняя вертикальная координата первого объекта (co1.y1) в промежутке между координатами y1 и y2 второго объекта, и если это так, возвращаем True (значит, первый координатный объект столкнулся со вторым верхней стороной).

Функция collided_bottom

Функция collided_bottom значительно отличается от предыдущих. Она проверяет столкновение нижней стороны первого координатного объекта со вторым:

```
   def collided_bottom(y, co1, co2):
❶      if within_x(co1, co2):
❷          y_calc = co1.y2 + y
❸          if y_calc >= co2.y1 and y_calc <= co2.y2:
❹              return True
❺      return False
```

Эта функция принимает дополнительный аргумент `y` — значение, которое мы прибавим к нижней *y*-координате первого объекта. В строке ❶ проверяем объекты на пересечение по горизонтали (аналогично коду функции `collided_top`). В строке ❷ прибавляем значение аргумента `y` к координате `y2` первого объекта и сохраняем результат в переменной `y_calc`. Если в строке ❸ новое значение `y_calc` окажется в промежутке между координатами `y1` и `y2` второго объекта, возвращаем `True` в строке ❹. Это соответствует столкновению нижней стороны `co1` с верхней стороной `co2`. Если предыдущие проверки не дали положительного результата, возвращаем `False` в строке ❺.

Поскольку человечек может упасть с платформы, необходим дополнительный аргумент — `y`. В отличие от остальных проверок на столкновение, в `collided_bottom` нужно определять не только уже произошедшее, но и будущее столкновение снизу (аргумент `y` позволит заглянуть вниз на заданное количество пикселей). Если человечек, сойдя с платформы, продолжит шагать по воздуху, игра получится не очень реалистичной. Поэтому проверим возможность его столкновения с платформой снизу, и, если проверка даст отрицательный результат, человечек будет стремительно падать!

Создаем класс Sprite

Класс-предок всех игровых объектов будет называться `Sprite` (спрайт), и в его составе будет две функции: `move` — для перемещения спрайта, и `coords`, возвращающая текущую позицию спрайта на игровом экране. Вот код класса `Sprite`:

```
   class Sprite:
❶      def __init__(self, game):
❷          self.game = game
❸          self.endgame = False
❹          self.coordinates = None
❺      def move(self):
❻          pass
❼      def coords(self):
❽         return self.coordinates
```

Определенная в строке ❶ функция __init__ класса `Sprite` принимает единственный аргумент — `game`. Это главный объект игры (принадлежащий классу `Game`, который мы создали в начале данной главы). Он нужен здесь, чтобы каждый спрайт, который мы создадим, имел доступ к списку остальных спрайтов в игре. В строке ❷ сохраняем аргумент `game` в одноименном свойстве.

End game — конец игры

В строке ❸ создаем свойство `endgame`, которое будет сигнализировать об окончании игры (пока присвоим ему значение `False`). В строке ❹ создаем последнее свойство, `coordinates` (координаты), со значением `None`, то есть «координаты не заданы».

Функция `move`, определенная в строке ❺, в этом классе-предке ничего не делает, поэтому вместо ее тела указываем в строке ❻ ключевое слово `pass`. А функция `coords`, определенная в строке ❼, в строке ❽ просто возвращает значение свойства `coordinates`.



Итак, в классе `Sprite` есть функция `move`, которая ничего не делает, и функция `coords`, которая возвращает пустое значение вместо координат. На первый взгляд, бесполезный код. Однако нам известно, что все классы-потомки класса `Sprite` унаследуют от него функции `move` и `coords`. Поэтому в главном цикле игры, где мы перебираем спрайты из списка, можно вызывать их функцию `move` — ведь она будет у каждого спрайта.

! *Классы с функциями, не выполняющими полезную работу, — обычное явление в программировании, своего рода соглашение о том, что все потомки будут предоставлять определенный набор функций, даже если в некоторых случаях это просто заглушки.*

Добавляем платформы

Теперь напишем код для платформ. Дадим классу объектов-платформ имя `PlatformSprite`, и он будет потомком класса `Sprite`. Функция `__init__` этого нового класса будет принимать аргумент `game` (аналогично своему предку — `Sprite`), а также изображение (`image`), позицию по горизонтали и вертикали (`x` и `y`), ширину изображения (`width`) и его высоту (`height`). Вот код класса `PlatformSprite`:

```
❶ class PlatformSprite(Sprite):
❷     def __init__(self, game, photo_image, x, y, width, height):
❸         Sprite.__init__(self, game)
❹         self.photo_image = photo_image
```

```
❺         self.image = game.canvas.create_image(x, y, \
                  image=self.photo_image, anchor='nw')
❻         self.coordinates = Coords(x, y, x + width, y + height)
```

Определяя класс PlatformSprite в строке ❶, указываем единственный аргумент — имя класса-предка (Sprite). Функция __init__ в строке ❷ принимает семь аргументов: self, game, photo_image, x, y, width и height.



В строке ❸ вызываем функцию __init__ класса-предка (Sprite), передавая ей в качестве аргументов self и game, поскольку именно эти аргументы и принимает функция __init__ класса Sprite.

На этом этапе новый объект класса PlatformSprite приобретет все свойства класса-предка (то есть свойства game, endgame и coordinates), потому что они создаются в функции __init__ класса Sprite, которую мы вызвали.

В строке ❹ сохраняем аргумент photo_image в свойстве с таким же именем, а в строке ❺ создаем изображение на экране (используя свойство canvas объекта game и функцию create_image) и сохраняем его идентификатор в свойстве image.

В строке ❻ создаем объект класса Coords, указывая значения x и y в качестве первых двух аргументов. Задаем вторую пару аргументов, прибавляя к x и y значения width и height соответственно.

Несмотря на то что свойство coordinates класса-предка (Sprite) получает сначала значение None, в классе-потомке PlatformSprite задаем ему новое значение — объект класса Coords, где содержится позиция платформы на игровом экране.

Создаем объект-платформу

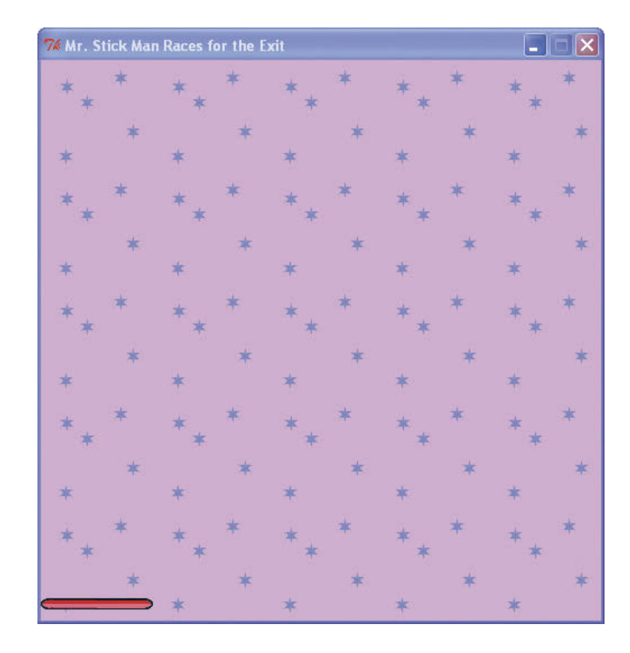
Добавим в игру единственную платформу, чтобы посмотреть, на что это похоже. Изменим последние строки файла с программой (*stickmangame.py*), чтобы они выглядели так:

```
❶ g = Game()
❷ platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
        0, 480, 100, 10)
❸ g.sprites.append(platform1)
❹ g.mainloop()
```

Строки ❶ и ❹ не изменились, но между ними возникли еще две строки: в строке ❷ создаем объект класса PlatformSprite, передавая ему главный объект игры (g), а также объект класса PhotoImage (который использует первое из созданных ранее изображений для платформ,

platform1.gif). Передаем позицию, в которой должна отображаться платформа (отступ 0 пикселей слева и 480 пикселей сверху — почти в самом низу холста), а также ширину и высоту изображения (100 пикселей и 10 пикселей). В строке ❸ добавляем только что созданный спрайт `platform1` в список спрайтов нашего объекта класса `Game` (`g`).

Если теперь запустить программу, в левой нижней части игрового экрана должна появиться платформа:



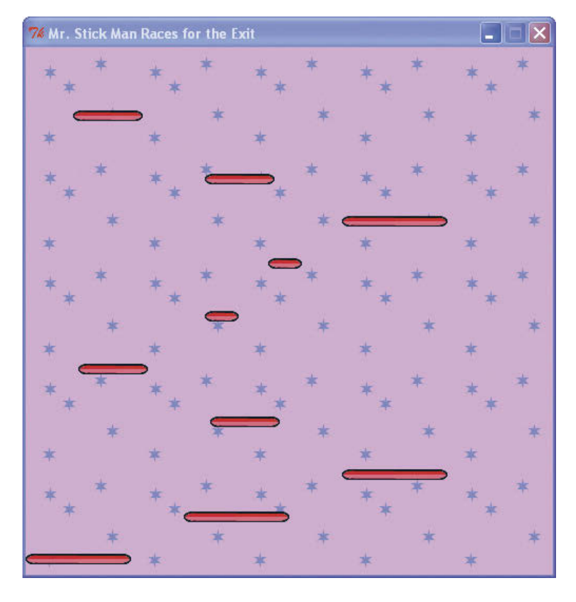
Больше платформ!

Теперь добавим в игру сразу десяток платформ. У каждой из них будет своя позиция по *x* и *y* — чтобы платформы распределились по всему экрану. Добавим в конец программы (вместо четырех строк из предыдущего раздела) такой код:

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    0, 480, 100, 10)
platform2 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    150, 440, 100, 10)
platform3 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 400, 100, 10)
platform4 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 160, 100, 10)
```

```
platform5 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    45, 60, 66, 10)
platform9 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
g.mainloop()
```

Мы создали десять объектов `PlatformSprite`, сохранив их в переменных `platform1`, `platform2`, `platform3` и так далее — до `platform10`. Затем поместили каждую платформу в список `sprites`, принадлежащий объекту `g` класса `Game`. Если теперь запустить игру, экран будет выглядеть так:



Итак, мы создали основу игры. Теперь можно добавить в нее главного героя — человечка.

Что мы узнали

В этой главе мы создали главный класс игры (Game) и заполнили экран фоновым изображением. Научились определять, пересекается ли пара горизонтальных или вертикальных координат с двумя другими горизонтальными или вертикальными координатами, создав для этого функции within_x и within_y. Использовали эти функции для проверки столкновения двух координатных объектов. Этот код пригодится в следующих главах, когда мы займемся анимацией человечка и нужно будет определять, не столкнулся ли он с платформой во время перемещения по игровому экрану.

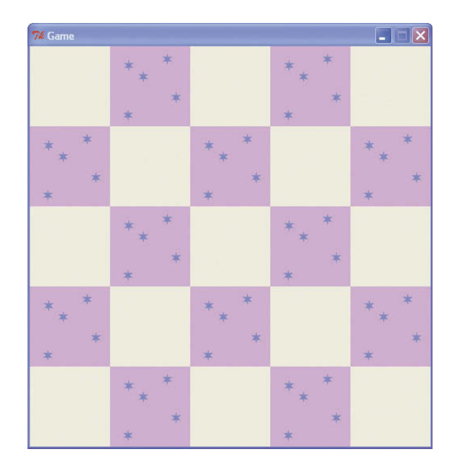
Также мы создали базовый класс Sprite и его первого потомка — класс PlatformSprite, которым воспользовались для отображения платформ на экране.

Упражнения

Поэкспериментируйте с фоновым изображением в игре, выполнив эти задания.

#1. Шахматная доска

Измените класс `Game` таким образом, чтобы фоновое изображение выглядело как шахматная доска:



#2. Шахматная доска с двумя изображениями

Выполнив предыдущее задание, доработайте код, чтобы для рисования доски использовались два чередующихся изображения. Создайте в графическом редакторе еще одни «обои» и измените класс `Game`, чтобы он рисовал не заполненные квадраты, чередующиеся с пустыми, а квадраты, заполненные разными изображениями.

#3. Книжная полка и лампа

Фон игры можно сделать интереснее, если в качестве обоев использовать разные изображения. Создайте несколько копий фоновых обоев и нарисуйте на одной копии книжную полку, на другой — стол с лампой и так далее. Затем измените код класса `Game`, чтобы там загружались (и вперемежку выводились на экран) несколько изображений (три или четыре).

17

СОЗДАЕМ ЧЕЛОВЕЧКА

В этой главе мы добавим в игру «Человечек спешит к выходу» главного героя. Код, который потребуется для этого написать, будет сложнее всего того, с чем мы до сих пор сталкивались. Ведь человечку нужно двигаться вправо и влево, прыгать, останавливаться при столкновении с платформой и падать, если он выбежит за ее пределы. Воспользуемся привязкой к событиям, чтобы фигурка на экране двигалась вправо-влево при нажатии соответствующих клавиш-стрелок и прыгала при нажатии клавиши «пробел».

Инициализация спрайта

Функция `__init__` класса, представляющего спрайт человечка, будет примерно такой же, как и у остальных классов в игре. Первым делом дадим новому классу (который также будет потомком класса `Sprite`) имя — `StickFigureSprite`.

```
class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
```

Этот код напоминает код класса `PlatformSprite` из главы 16, однако здесь мы не передаем в функцию `__init__` другие аргументы, кроме `self` и `game`, ведь в отличие от множества платформ, объект класса `StickFigureSprite` будет в игре один.

Загрузка изображений фигурки



Поскольку платформ на игровом экране много и каждой может соответствовать изображение произвольного размера, мы передавали то или иное изображение в функцию `__init__` класса `PlatformSprite`. Однако фигурка человечка в игре будет одна, а значит, ни к чему загружать ее изображения вне класса и при создании объекта передавать их через аргументы. Класс `StickFigureSprite` будет загружать необходимые изображения сам.

Для этого и предназначены несколько следующих строк функции `__init__`. В них загружаются три изображения фигурки, развернутой влево, и три изображения фигурки, развернутой вправо. Их нужно загрузить именно при создании объекта, чтобы не делать это снова и снова при каждом выводе фигурки на экран (иначе наша программа будет слишком медленной). Вот эти строки:

```
    class StickFigureSprite(Sprite):
        def __init__(self, game):
            Sprite.__init__(self, game)
❶           self.images_left = [
                PhotoImage(file="figure-L1.gif"),
                PhotoImage(file="figure-L2.gif"),
                PhotoImage(file="figure-L3.gif")
            ]
❷           self.images_right = [
                PhotoImage(file="figure-R1.gif"),
                PhotoImage(file="figure-R2.gif"),
                PhotoImage(file="figure-R3.gif")
            ]
❸           self.image = game.canvas.create_image(200, 470, \
                    image=self.images_left[0], anchor='nw')
```

Загружаем три изображения, которые нужны для анимации фигурки, бегущей влево, и три изображения для анимации фигурки, бегущей вправо.

В строках ❶ и ❷ создаем свойства `images_left` и `images_right`. Каждое их них содержит список объектов класса `PhotoImage`, соответствующих изображениям, которые мы создали в главе 15.

В строке ❸ с помощью функции `create_image` выводим на экран первое изображение (`images_left[0]`) в координатах (200, 470). Это середина холста по горизонтали и его нижняя часть по вертикали.

Функция create_image возвращает идентификатор, который мы сохраняем в свойстве image (он понадобится нам позже).

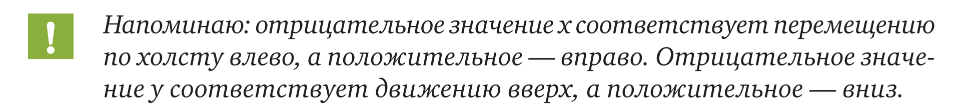
Настройка свойств

В следующих строках функции __init__ создадим еще несколько свойств.

```
        self.images_right = [
            PhotoImage(file="figure-R1.gif"),
            PhotoImage(file="figure-R2.gif"),
            PhotoImage(file="figure-R3.gif")
        ]
        self.image = game.canvas.create_image(200, 470, \
            image=self.images_left[0], anchor='nw')
❶       self.x = -2
❷       self.y = 0
❸       self.current_image = 0
❹       self.current_image_add = 1
❺       self.jump_count = 0
❻       self.last_time = time.time()
❼       self.coordinates = Coords()
```

Свойства x и y, созданные в строках ❶ и ❷, предназначены для хранения величин, на которые мы будем смещать фигурку по горизонтали (координаты *x1* и *x2*) и вертикали (координаты *y1* и *y2*), то есть двигать ее по холсту.

Как мы знаем из главы 13, чтобы создать эффект движения с помощью tkinter, нужно периодически смещать объект относительно его текущих координат. Присваиваем свойству x значение –2, а свойству y — 0. Таким образом, будем вычитать 2 из позиции по *x*, а вертикальную позицию оставим неизменной, в результате чего сразу после запуска игры фигурка будет двигаться влево.



Напоминаю: отрицательное значение x соответствует перемещению по холсту влево, а положительное — вправо. Отрицательное значение y соответствует движению вверх, а положительное — вниз.

Current image — текущее изображение

В строке ❸ создаем свойство current_image, чтобы хранить там индекс текущего изображения. Список кадров анимации для движения влево (images_left) содержит изображения *figure-L1.gif*, *figure-L2.gif* и *figure-L3.gif*, которым соответствуют индексы 0, 1 и 2.

В строке ❹ создаем свойство current_image_add. Оно содержит число, которое нужно прибавить к индексу, хранящемуся в свойстве

current_image, чтобы получить индекс следующего изображения. Например, если сейчас на экране изображение с индексом 0, то нужно прибавить 1, и мы получим индекс следующего кадра ❶, а затем — еще 1 для отображения индекса последнего кадра в списке ❷. В следующей главе разберемся, как использовать это свойство.

Current image add — добавить текущее изображение

В строке ❺ создаем свойство jump_count. Это счетчик, который понадобится для прыжков человечка. А в свойстве last_time, которое мы создаем в строке ❻, будем хранить время последней смены кадров фигурки. Сейчас мы записываем туда текущее время с помощью функции time из модуля time.

Jump count — счет прыжков

Last time — в последний раз

В строке ❼ сохраняем в свойстве coordinates объект класса Coords, созданный без передачи аргументов (то есть значения x1, y1, x2 и y2 равны 0). В отличие от спрайтов платформ координаты фигурки будут меняться, так что мы зададим эти значения позже.

Привязка к нажатиям клавиш

В последних строках функции __init__ с помощью функции bind_all привяжем нажатие нужных нам клавиш к вызовам соответствующих функций.

Bind all — привязать всё

```
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)
```

Чтобы фигуркой можно было управлять, необходимо создать функции turn_left (повернуть налево), turn_right (повернуть направо) и jump (прыгать). Привязываем клавишу-стрелку «влево» (<KeyPress-Left>) к вызову функции turn_left, клавишу-стрелку «вправо» (<KeyPress-Right>) — к вызову функции turn_right, а клавишу «пробел» (<space>) — к функции jump.



Поворот фигурки вправо и влево

Функции turn_left и turn_right проверяют, что фигурка не находится в прыжке и не падает (в нашей игре нельзя менять направление во время прыжка), и если это так, задают свойству x значение, которое соответствует движению влево (turn_left) или вправо (turn_right):

```
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

❶   def turn_left(self, evt):
❷       if self.y == 0:
❸           self.x = -2

❹   def turn_right(self, evt):
❺       if self.y == 0:
❻           self.x = 2
```

При нажатии клавиши-стрелки «влево» Python будет вызывать функцию `turn_left`, передавая в нее объект, содержащий информацию о событии (аргумент `evt`).

! *Для наших целей объект события не нужен, однако соответствующий ему аргумент должен быть у обеих функций (строки ❶ и ❹). Иначе Python, попытавшись передать этот аргумент, выдаст ошибку. Объект события содержит x и y-координаты курсора мыши (для событий мыши), код нажатой клавиши (для событий клавиатуры) и прочую информацию. В нашей игре эти сведения не пригодятся, поэтому на аргумент evt можно не обращать внимания.*

Убедиться в том, что фигурка не находится в прыжке, поможет проверка свойства `y` в строках ❷ и ❺. Прыжку или падению соответствует его ненулевое значение. Если *y* равен 0, задаем свойству *x* значение –2 (для поворота влево, строка ❸) или 2 (для поворота вправо, строка ❻). Можно было бы использовать числа 1 и –1, но тогда фигурка будет двигаться слишком медленно. (Чтобы оценить разницу, попробуйте поменять эти значения, когда мы закончим анимацию человечка.)

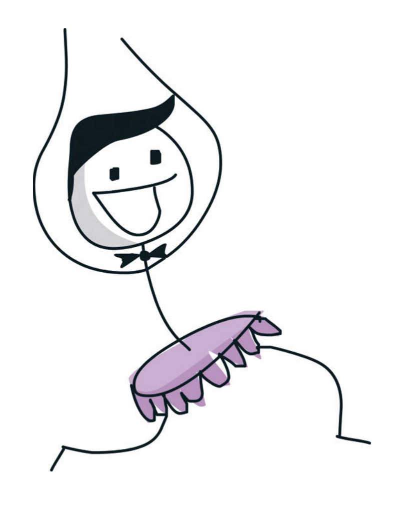
Прыжок фигурки

Jump — прыгнуть

Функция для прыжка называется `jump`, она очень похожа на функции `turn_left` и `turn_right`.

```
    def turn_right(self, evt):
        if self.y == 0:
            self.x = 2

    def jump(self, evt):
❶       if self.y == 0:
❷           self.y = -4
❸           self.jump_count = 0
```



Эта функция принимает аргумент evt (объект события), на который мы не будем обращать внимания, поскольку нам не нужна дополнительная информация. И так ясно, что если эта функция вызвана, значит, нажата клавиша «пробел».

Чтобы в прыжке нельзя было прыгнуть еще раз, в строке ❶ проверяем свойство y на равенство нулю. Если это так (фигурка не находится в прыжке и не падает), в строке ❷ задаем свойству y значение –4 (чтобы фигурка двигалась вверх), а в строке ❸ обнуляем свойство jump_count (счетчик ограничения длительности прыжка). Прыжок не должен продолжаться бесконечно, поэтому фигурка будет подниматься вверх на заданный счет, а затем опускаться, словно под действием силы тяжести. Код для этого мы напишем в следующей главе.

Что мы уже написали

Вспомним все классы и функции, которые есть сейчас в игре, и убедимся, что в файле с программой они расположены правильно.

Вверху программы должны быть команды импортирования модулей, а следом за ними — классы Game и Coords. Класс Game будет главным управляющим классом игры, а объекты класса Coords используем для хранения позиций графических объектов (платформ и человечка):

```
from tkinter import *
import random
import time

class Game:
    ...
class Coords:
    ...
```

Дальше идут функции проверки столкновений спрайтов (их имена начинаются с «within» и «collided»), базовый класс Sprite (предок всех спрайтов), класс PlatformSprite (с его помощью мы создаем спрайты-платформы) и незавершенный класс StickFigureSprite (который соответствует спрайту человечка):

```
def within_x(co1, co2):
    ...
def within_y(co1, co2):
    ...
```

```
def collided_left(co1, co2):
    ...
def collided_right(co1, co2):
    ...
def collided_top(co1, co2):
    ...
def collided_bottom(y, co1, co2):
    ...
class Sprite:
    ...
class PlatformSprite(Sprite):
    ...
class StickFigureSprite(Sprite):
    ...
```

В конце программы должен располагаться код, создающий все объекты, которые есть в игре на данный момент: главный объект класса `Game` и объекты-платформы. В последней строке кода мы вызываем функцию `mainloop` главного объекта:

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    0, 480, 100, 10)
...
g.sprites.append(platform1)
...
g.mainloop()
```

Если ваш код выглядит иначе или работает неправильно, загляните в конец главы 18, где целиком напечатан код игры.

Что мы узнали

В этой главе мы начали писать класс для фигурки человечка. Если сейчас создать объект этого класса, толку от него будет мало. Он лишь загрузит нужные для анимации изображения и создаст несколько свойств, которые понадобятся нам позже. Однако в этом классе уже есть функции для изменения значений свойств при возникновении событий клавиатуры (если игрок нажмет клавишу-стрелку «влево», «вправо» или клавишу «пробел»).

В следующей главе мы закончим работу над игрой. Добавим в класс `StickFigureSprite` функции для отображения человечка на холсте, а также для его анимации и перемещения по игровому экрану. Добавим в игру дверь — выход, до которого человечку нужно добраться.

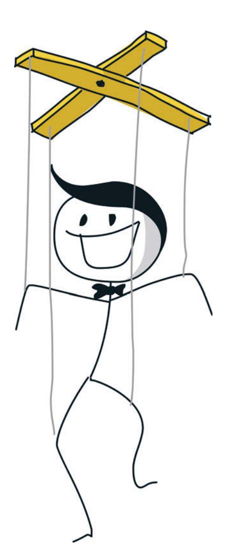
18

ДОДЕЛЫВАЕМ ИГРУ

В трех предыдущих главах мы занимались игрой «Человечек спешит к выходу». Создали графику, написали код для вывода фонового изображения, платформ и фигурки человечка. В этой главе разработаем фрагменты кода для анимации фигурки, а также добавим в игру дверь.

Если вы запутаетесь при написании программы, сравните ваш код с окончательным кодом игры, который напечатан в конце этой главы.

Анимация фигурки



У нас уже есть класс для фигурки, который загружает нужные изображения и привязывает нажатие клавиш к некоторым функциям. Однако это еще не всё. Запустив игру сейчас, ничего интересного мы не увидим.

Пора добавить в класс `StickFigureSprite` функции `animate`, `move` и `coords`. Функция `animate` будет отображать различные кадры анимации фигурки, `move` — определять, куда фигурка должна переместиться, а `coords` — возвращать ее текущую позицию (в отличие от платформ, позицию человечка нужно будет пересчитывать по мере его движения по экрану).

Создаем функцию animate

Напишем функцию `animate`, которая будет менять кадры анимации фигурки в зависимости от того, куда она движется.

Проверка движения

Нам не нужно, чтобы кадры анимации фигурки сменялись слишком часто, иначе ее движение будет выглядеть неправдоподобно. Вспомните о рисованной анимации на страницах блокнота. Если чересчур быстро перелистывать странички, сложно разобрать, что происходит.

Введем код функции animate по частям. В первой части кода будем проверять, куда движется фигурка — влево или вправо, затем с помощью свойства last_time (в последний раз) определим, пора ли переходить к следующему кадру анимации (свойство last_time помогает контролировать скорость чередования кадров). Располагаться функция animate будет после функции jump, которую мы создали в главе 17.

```
    def jump(self, evt):
        if self.y == 0:
            self.y = -4
            self.jump_count = 0

    def animate(self):
❶       if self.x != 0 and self.y == 0:
❷           if time.time() - self.last_time > 0.1:
❸               self.last_time = time.time()
❹               self.current_image += self.current_image_add
❺               if self.current_image >= 2:
❻                   self.current_image_add = -1
❼               if self.current_image <= 0:
❽                   self.current_image_add = 1
```

В строке ❶ проверяем, что x не равно 0 (что фигурка движется влево или вправо), а y равно 0 (фигурка не в прыжке и не падает). Если оба условия выполняются, фигурку нужно анимировать, если нет, изображение должно остаться прежним и следующие строки кода будут пропущены.

В строке ❷ вычисляем, сколько времени прошло с предыдущего вызова функции animate. Для этого вычитаем значение свойства last_time из текущего времени, которое возвращает функция time.time(). Так мы узнаем, пора ли выводить следующий кадр анимации. Если результат превышает десятую долю секунды (0.1), переходим к блоку кода, который начинается со строки ❸. Сохраняем в свойстве last_time текущее время, сбрасывая таким образом таймер, чтобы начать отсчет до следующей смены кадров.

В строке ❹ прибавляем значение свойства current_image_add к значению свойства current_image, где хранится индекс текущего кадра. Напомню, что свойство current_image_add создается в функции __init__ (см. главу 17) и при первом вызове функции animate его значение равно 1.

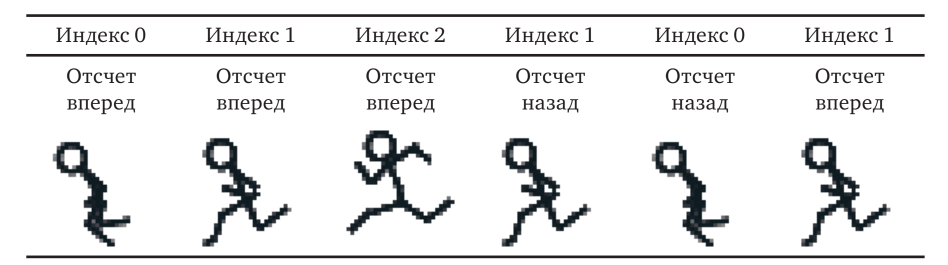
В строке ❺ стоит проверка: если значение индекса, который хранится в свойстве `current_image`, больше или равно 2, в строке ❻ меняем значение `current_image_add` на –1. Подобная проверка выполняется и в строке ❼: если значение `current_image` достигло 0, нужно снова увеличивать индекс, для этого в строке ❽ присваиваем `current_image_add` значение 1.

> **!** *Если возникли сложности с расстановкой отступов в этом коде, вот подсказка: в начале строки ❶ стоит 8 пробелов, а в начале строки ❽ — 20 пробелов.*

Чтобы лучше понять логику этой функции, представьте набор цветных кубиков, лежащих в ряд на полу. Вы двигаете палец от кубика к кубику, при этом каждому кубику соответствует порядковый номер (значение свойства `current_image`). За один раз ваш палец перемещается на столько кубиков вперед или назад, какое число хранится в свойстве `current_image_add`. Если там 1, палец передвинется на один кубик вперед, а если –1 — то на один кубик назад. Номер следующего кубика, на который укажет ваш палец, можно узнать, прибавляя к `current_image` значение `current_image_add`.

Это и происходит в коде функции `animate`, только вместо цветных кубиков — по три кадра анимации фигурки для каждого направления. Они хранятся в двух списках, индексы этих кадров — 0, 1 и 2. Дойдя до последнего кадра, начинаем обратный отсчет, а дойдя до первого, переходим на отсчет по возрастанию индексов. Возникает эффект бегущей фигурки.

На схеме показано, как мы перебираем кадры из списка с помощью индексов, которые вычисляются в функции `animate`.



Индекс 0	Индекс 1	Индекс 2	Индекс 1	Индекс 0	Индекс 1
Отсчет вперед	Отсчет вперед	Отсчет вперед	Отсчет назад	Отсчет назад	Отсчет вперед

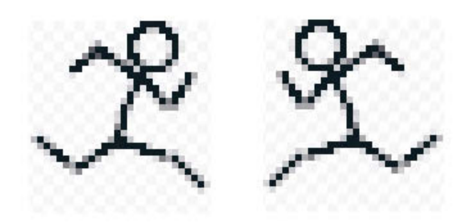
Смена кадра

Вторая часть функции `animate` предназначена для смены текущего кадра в соответствии с вычисленным ранее индексом:

```
    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
```

```
                self.last_time= time.time()
                self.current_image += self.current_image_add
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
❶       if self.x < 0:
❷           if self.y != 0:
❸               self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
❹           else:
❺               self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[self.current_image])
❻       elif self.x > 0:
❼           if self.y != 0:
❽               self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
❾           else:
❿               self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[self.current_image])
```

В строке ❶ выполняется проверка: если свойство x меньше 0, значит, фигурка движется влево, в этом случае мы переходим к блоку кода со строки ❷ по строку ❺. В строке ❷ сравниваем свойство y с 0, и если y не равно 0 (фигурка находится в прыжке или падает), в строке ❸ с помощью функции itemconfig меняем изображение фигурки на последний кадр в списке изображений, повернутых влево (images_left[2]). Выбираем кадр с самым широким шагом, чтобы анимация прыжка выглядела более реалистично:



Если y равно 0, в строке ❹ выполняется вариант else, а в строке ❺ с помощью itemconfig меняем текущее изображение на кадр, индекс которого находится в свойстве current_image.

В строке ❻ проверяем, движется ли фигурка вправо (x больше 0). Если это так, выполняется блок кода со строки ❼ по строку ❿. Этот код очень похож на код для движения влево: делается проверка, находится ли фигурка в прыжке, и так далее, но кадры выбираются из списка images_right, где хранятся изображения фигурки, повернутой вправо.

Получение позиции фигурки

Нам важно знать, где находится фигурка на игровом экране, а поскольку она движется, функция coords класса StickFigureSprite будет отличаться от одноименной функции родительского класса Sprite. Воспользуемся функцией холста coords, чтобы узнать положение фигурки, и в соответствии с этим выставим значения x1, y1, x2 и y2 свойства coordinates, которое создается в функции __init__ класса Sprite. Добавьте этот код после функции animate:

```
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[self.current_image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[self.current_image])

    def coords(self):
❶       xy = self.game.canvas.coords(self.image)
❷       self.coordinates.x1 = xy[0]
❸       self.coordinates.y1 = xy[1]
❹       self.coordinates.x2 = xy[0] + 27
❺       self.coordinates.y2 = xy[1] + 30
        return self.coordinates
```

Создавая в главе 16 класс Game, мы добавили в него свойство canvas, содержащее объект-холст. В строке ❶ вызываем функцию coords этого холста (self.game.canvas.coords), которая возвращает *x*- и *y*-координаты изображения (идентификатор изображения хранится в свойстве image, которое мы передаем функции в качестве аргумента).

Функция холста coords возвращает список (мы сохраняем его в переменной xy), где находятся два элемента — *x*- и *y*-координаты левого верхнего угла изображения. В строке ❷ сохраняем *x*-координату как значение x1 свойства coordinates, а в строке ❸ — *y*-координату как значение y1 свойства coordinates.

Поскольку все изображения фигурки одинакового размера — 27 пикселей в ширину и 30 в высоту, — мы можем вычислить значения x2 и y2, прибавляя ширину (в строке ❹) и высоту (в строке ❺) к координатам *x* и *y* соответственно.

В последней строке кода возвращаем из функции свойство coordinates.

Перемещение фигурки по экрану

Последняя из функций класса StickFigureSprite — move — отвечает за перемещение человечка по холсту, а также обрабатывает столкновения фигурки с другими объектами или границами экрана.

Начало функции move

Вот первая часть функции move. Этот код должен располагаться после функции coords:

```
    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates

    def move(self):
❶       self.animate()
❷       if self.y < 0:
❸           self.jump_count += 1
❹           if self.jump_count > 20:
❺               self.y = 4
❻       if self.y > 0:
❼           self.jump_count -= 1
```

В строке ❶ вызываем функцию animate, созданную ранее (она переключает кадры анимации по мере необходимости). В строке ❷ проверяем, меньше ли нуля значение y. Если да, значит фигурка находится в прыжке, поскольку отрицательное значение y соответствует движению вверх (напоминаю: верх холста — это *y*-координата 0, а низ — *y*-координата 500).

В строке ❸ увеличиваем jump_count на 1, а в строке ❹ проверяем, не стало ли значение jump_count больше 20. Если стало, в строке ❺ задаем свойству y значение 4, чтобы фигурка начала падать.

В строке ❻ проверяем: y больше 0? Если так, значит, фигурка падает, тогда мы уменьшаем jump_count на 1, поскольку, досчитав до 20, нужно переходить на обратный отсчет. Чтобы представить, как работает счетчик jump_count, медленно поднимайте руку вверх, считая до 20, а затем опускайте ее вниз, считая от 20 до 0.



В нескольких следующих строках функции move вызываем функцию coords, которая возвращает позицию фигурки на экране, и сохраняем эту позицию в переменной co. Затем создаем переменные left, right, top, bottom и falling, которые нам скоро понадобятся.

Falling — падающий

```
        if self.y > 0:
            self.jump_count -= 1
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
```

Эти переменные (вначале они равны True) будут контролировать, нужно ли проверять фигурку на столкновение с каждой из четырех сторон и на падение.

Следующий фрагмент функции move проверяет, не столкнулась ли фигурка с верхней или нижней границей холста:

```
        bottom = True
        falling = True
❶       if self.y > 0 and co.y2 >= self.game.canvas_height:
❷           self.y = 0
❸           bottom = False
❹       elif self.y < 0 and co.y1 <= 0:
❺           self.y = 0
❻           top = False
```

Когда фигурка падает (y больше 0), нужно проверять ее на столкновение с нижней границей холста, и если это произошло, останавливать падение, иначе человечек вылетит за пределы игрового экрана. Для этого

в строке ❶ проверяем, верно ли, что значение y2 (низ фигурки) больше или равно свойству canvas_height (высота холста) объекта game. Если это так, в строке ❷ обнуляем свойство y, чтобы остановить падение, а затем в строке ❸ присваиваем переменной bottom значение False. Для остальной части кода это будет признаком, что проверять фигурку на столкновение снизу больше не требуется.

Обработка столкновения фигурки с верхней границей холста выполняется примерно так же. В строке ❹ проверяем, находится ли фигурка в прыжке (y меньше 0) и столкнулась ли она с верхом холста (координата y1 меньше или равна 0). Если обе проверки дают положительный результат, в строке ❺ обнуляем y, чтобы остановить движение вверх, а в строке ❻ присваиваем переменной top значение False. Это будет признаком того, что проверять фигурку на столкновение сверху больше не нужно.

Практически так же мы будем проверять, столкнулась ли фигурка с левой или правой границей холста:

```
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            top = False
❶       if self.x > 0 and co.x2 >= self.game.canvas_width:
❷           self.x = 0
❸           right = False
❹       elif self.x < 0 and co.x1 <= 0:
❺           self.x = 0
❻           left = False
```

В строке ❶ проверяем, движется ли фигурка вправо (x больше 0) и столкнулась ли она с правой границей холста (координата x2 больше или равна ширине холста, которая хранится в свойстве canvas_width объекта game). Если оба условия выполняются, в строке ❷ обнуляем x, чтобы остановить горизонтальное движение, а в строке ❸ присваиваем переменной right значение False, чтобы больше не обрабатывать столкновение справа. В строках с ❹ по ❻ аналогичные проверки выполняются и для левой границы холста.

Столкновение с другими спрайтами

Разобравшись с границами игрового экрана, выясним, не столкнулась ли фигурка с другим игровым объектом. Создадим цикл, чтобы рассмотреть все спрайты из списка sprites объекта game и проверить фигурку на столкновение с каждым из них.

```
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            left = False
```

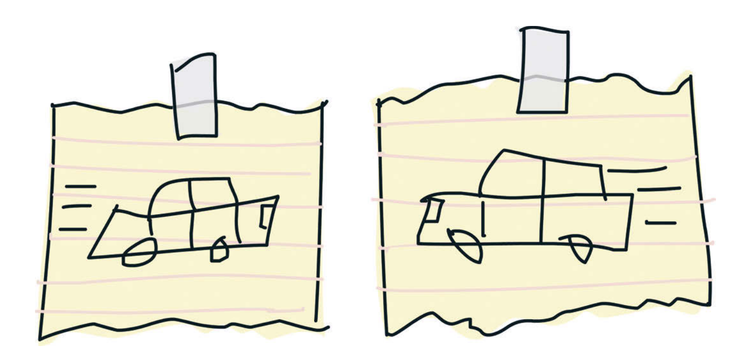
```
❶     for sprite in self.game.sprites:
❷         if sprite == self:
❸             continue
❹         sprite_co = sprite.coords()
❺         if top and self.y < 0 and collided_top(co, sprite_co):
❻             self.y = -self.y
❼             top = False
```

Continue — продолжить

В строке ❶ в цикле перебираем элементы из списка sprites. На каждом шаге переменная sprite будет соответствовать очередному спрайту. В строке ❷ проверяем sprite на равенство self (self соответствует спрайту самого Человечка). Столкновение с самим собой проверять не нужно, поэтому, если sprite равняется self, в строке ❸ с помощью continue переходим к следующему повтору цикла (и к следующему спрайту).

Получаем координаты очередного спрайта, вызывая его функцию coords в строке ❹ и сохраняя результат в переменной sprite_co. Чтобы обработать случай, когда фигурка сталкивается со спрайтом верхней стороной, в строке ❺ проверяем следующие условия:

- Фигурка не столкнулась с верхней границей холста (переменная top все еще равна True).
- Фигурка находится в прыжке (значение y меньше 0).
- Фигурка столкнулась с текущим спрайтом из списка своей верхней стороной (для этой проверки используем функцию collided_top, созданную в главе 16).



Если все условия выполняются, нужно, чтобы фигурка начала падать. В строке ❻ с помощью операции минус (–) меняем значение y на противоположное. В строке ❼ присваиваем переменной top значение False, чтобы больше не проверять фигурку на столкновение верхней стороной.

Столкновение нижней стороной

Следующий фрагмент кода находится внутри того же цикла и служит для обработки столкновения фигурки нижней стороной с каким-либо из спрайтов:

```
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
❶           if bottom and self.y > 0 and collided_bottom(self.y, \
                    co, sprite_co):
❷               self.y = sprite_co.y1 - co.y2
❸               if self.y < 0:
❹                   self.y = 0
❺               bottom = False
❻               top = False
```

В строке ❶ выполняется три проверки: равна ли `True` переменная `bottom`, падает ли сейчас фигурка (`y` больше 0) и столкнулась ли фигурка со спрайтом нижней стороной. Если все условия выполняются, в строке ❷ помещаем в свойство `y` разницу между верхней *y*-координатой спрайта из списка (`y1`) и нижней *y*-координатой фигурки (`y2`). Это может показаться странным, так что давайте разберемся, почему мы действуем именно так.

Представьте, что человечек упал с платформы. Он движется вниз по экрану, опускаясь на 4 пикселя при каждом повторе цикла в функции `mainloop`, и вот низ фигурки оказывается на 3 пикселя выше еще одной платформы. Предположим, низу фигурки (`y2`) соответствует *y*-координата 57, а верху платформы (`y1`) — *y*-координата 60. Тогда функция `collided_bottom` вернет `True`, поскольку сумма свойства `y` (это 4) и координаты фигурки `y2` (это 57) даст 61.

Нам не нужно, чтобы человечек тут же остановился, словно уже ударился о платформу или о низ игрового экрана. Это будет выглядеть так, будто он прыгнул вниз и завис в воздухе, немного не долетев до земли. Лучше вычтем координату `y2` фигурки (это 57) из координаты `y1` платформы (это 60), получим 3. Присвоим это значение свойству `y` (тогда все будет в порядке, ведь как раз на столько пикселей должна опуститься фигурка, чтобы встать на платформу).

В строке ❸ проверяем, не получилось ли в результате вычислений отрицательное число. Если получилось, обнуляем `y` в строке ❹, чтобы фигурка не начала снова подниматься вверх.

В строках ❺ и ❻ присваиваем переменным `bottom` и `top` значение `False`, чтобы больше не проверять фигурку на столкновение верхней и нижней сторонами с другими спрайтами.

Теперь добавим еще одну проверку — для обработки ситуации, когда фигурка находится на платформе и может выбежать за ее край:

```
            if self.y < 0:
                self.y = 0
            bottom = False
            top = False
        if bottom and falling and self.y == 0 \
                and co.y2 < self.game.canvas_height \
                and collided_bottom(1, co, sprite_co):
            falling = False
```

Для установки переменной `falling` в `False` должны выполняться пять условий:

1. Переменная `bottom` равна `True`;
2. Переменная `falling` равна `True` (проверка на падение);
3. В данный момент фигурка не падает (`y` равно 0);
4. Фигурка не столкнулась нижней стороной с нижней границей холста (`y2` меньше, чем `canvas_height`);
5. Низ фигурки соприкасается с верхом платформы (функция `collided_bottom` возвращает `True`).

Если все условия выполняются, присваиваем переменной `falling` значение `False`.

Столкновение слева и справа

Мы уже разобрались со столкновениями сверху и снизу, теперь нужно проверить, не столкнулась ли фигурка с чем-нибудь слева и справа:

```
        if bottom and falling and self.y == 0 \
                and co.y2 < self.game.canvas_height \
                and collided_bottom(1, co, sprite_co):
            falling = False
❶       if left and self.x < 0 and collided_left(co, sprite_co):
❷               self.x = 0
❸               left = False
❹       if right and self.x > 0 and collided_right(co, sprite_co):
❺               self.x = 0
❻               right = False
```

В строке ❶ смотрим, нужно ли обрабатывать столкновения слева (`left` все еще равняется `True`) и движется ли фигурка влево (`x` меньше 0). Вызывая функцию `collided_left`, проверяем столкновение левой стороны

фигурки со спрайтом. Если все в порядке, в строке ❷ обнуляем `x` (чтобы остановить фигурку), а в строке ❸ присваиваем переменной `left` значение `False`, чтобы больше не выполнять проверок на столкновение слева.

Столкновение справа в строке ❹ обрабатывается аналогичным образом. При выполнении всех условий обнуляем `x` в строке ❺, устанавливаем переменную `right` в `False` в строке ❻ и больше к этому не возвращаемся.

Полный цикл проверок (с четырех сторон) должен выглядеть так:

```
elif self.x < 0 and co.x1 <= 0:
    self.x = 0
    left = False
for sprite in self.game.sprites:
    if sprite == self:
        continue
    sprite_co = sprite.coords()
    if top and self.y < 0 and collided_top(co, sprite_co):
        self.y = -self.y
        top = False
    if bottom and self.y > 0 and collided_bottom(self.y, \
            co, sprite_co):
        self.y = sprite_co.y1 - co.y2
        if self.y < 0:
            self.y = 0
        bottom = False
        top = False
    if bottom and falling and self.y == 0 \
            and co.y2 < self.game.canvas_height \
            and collided_bottom(1, co, sprite_co):
        falling = False
    if left and self.x < 0 and collided_left(co, sprite_co):
        self.x = 0
        left = False
    if right and self.x > 0 and collided_right(co, sprite_co):
        self.x = 0
        right = False
```

Осталось добавить в функцию `move` несколько строк (уже после цикла):

```
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
❶       if falling and bottom and self.y == 0 \
                and co.y2 < self.game.canvas_height:
❷           self.y = 4
❸       self.game.canvas.move(self.image, self.x, self.y)
```

В строке ❶ проверяем переменные `falling` и `bottom` на равенство `True` (если так, значит, перебрав все спрайты в списке, мы не обнаружили столкновений снизу). Затем еще одна проверка: нижняя *y*-координата фигурки меньше высоты холста? Если так, фигурка находится над землей (над нижней границей холста). Поскольку человечек ни с чем не соприкасается снизу и находится выше уровня земли, значит, он завис в воздухе и ему остается только падать, потому что он выбежал за край платформы. Чтобы этого не произошло, в строке ❷ присваиваем свойству `y` значение 4.

В строке ❸ двигаем изображение человечка по экрану в соответствии с текущими значениями свойств `x` и `y`. Во время обработки столкновений эти свойства могли обнулиться — тогда фигурка останется на прежнем месте.

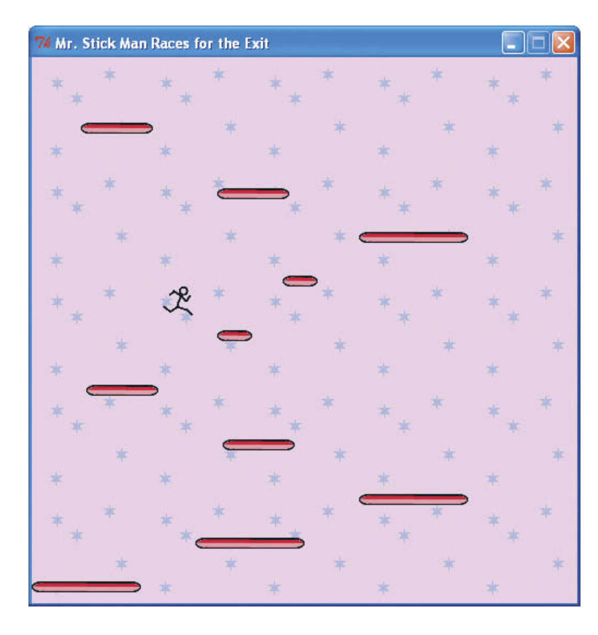
Проверяем спрайт человечка

Класс `StickFigureSprite` готов к использованию. Проверим, как он работает, добавив две строки кода перед вызовом функции `mainloop`:

```
❶ sf = StickFigureSprite(g)
❷ g.sprites.append(sf)
  g.mainloop()
```

В строке ❶ создаем объект класса `StickFigureSprite`, сохраняя его в переменной `sf`. В строке ❷ добавляем этот объект в список спрайтов так же, как делали это с платформами.

Запустите программу и убедитесь, что человечек может бегать, прыгать с платформы на платформу и падать.



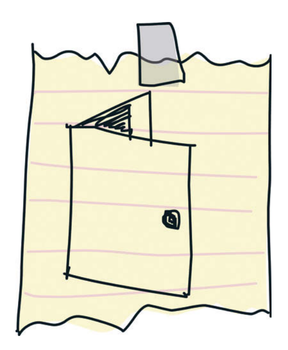
Дверь

Не хватает одного элемента — выхода, то есть двери. Создадим спрайт двери, напишем код, позволяющий человечку в нее войти, и добавим в программу объект-дверь.

Создаем класс DoorSprite

Для двери нужен еще один класс. Назовем его DoorSprite. Вот первая часть кода:

```
   class DoorSprite(Sprite):
❶      def __init__(self, game, photo_image, x, y, width, height):
❷          Sprite.__init__(self, game)
❸          self.photo_image = photo_image
❹          self.image = game.canvas.create_image(x, y, \
                   image=self.photo_image, anchor='nw')
❺          self.coordinates = Coords(x, y, x + (width / 2), y + height)
❻          self.endgame = True
```



В строке ❶ определяем функцию __init__ класса DoorSprite, которая принимает семь аргументов: self, game (главный объект игры), photo_image (изображение), координаты x и y, width (ширина изображения) и height (высота изображения). В строке ❷ вызываем функцию __init__ класса Sprite (так же, как вызывали ее в других классах, для которых Sprite является предком).

В строке ❸ сохраняем аргумент photo_image в одноименном свойстве (так же, как делали это для класса PlatformSprite). В строке ❹ создаем изображение на холсте с помощью функции create_image и сохраняем полученный из этой функции идентификатор в свойстве image.

В строке ❺ выставляем значение свойства coordinates, используя аргументы x и y в качестве координат *x1* и *y1* и вычисляя значения координат *x2* и *y2*. Чтобы получить *x2*, прибавляем половину ширины изображения к аргументу x. Например, если x равно 10 (соответственно, координата *x1* тоже равна 10), а ширина (width) — 40 пикселям, координата *x2* примет значение 30 (10 + 40/2).

Эти вычисления нужны, чтобы человечек не уперся в дверь сбоку, а вошел в нее. Запустив игру и добравшись до двери, вы увидите, что получилось.

Координату *y2* вычислять проще, чем *x2*. Достаточно прибавить высоту изображения (height) к аргументу y.

В строке ❻ задаем свойству endgame значение True. Это признак того, что фигурка добежала до двери и игра закончилась.

Дверь и конец игры

Теперь нужно изменить код функции move класса StickFigureSprite — ту его часть, которая обрабатывает столкновения со спрайтами справа и слева. Вот первое изменение:

```
        if left and self.x < 0 and collided_left(co, sprite_co):
            self.x = 0
            left = False
            if sprite.endgame:
                self.game.running = False
```

Проверяем, содержит ли свойство endgame спрайта, с которым фигурка столкнулась левой стороной, True. Если это так, даем свойству running значение False (фигурка добралась до двери, и игру нужно завершить).

Добавим в код, проверяющий столкновение справа, эти же строки:

```
        if right and self.x > 0 and collided_right(co, sprite_co):
            self.x = 0
            right = False
            if sprite.endgame:
                self.game.running = False
```

Добавляем объект-дверь

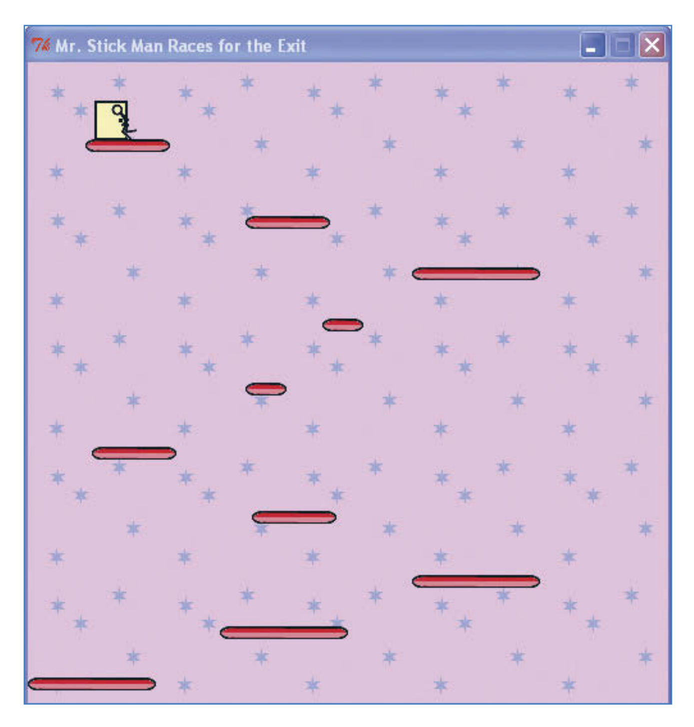
Осталось добавить в игру объект класса DoorSprite, то есть дверь. Сделать это нужно до вызова функции mainloop. Создадим объект-дверь раньше, чем объект-фигурку, и добавим ее в список спрайтов:

```
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, PhotoImage(file="door1.gif"), 45, 30, 40, 35)
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```

При создании класса DoorSprite передаем следующие аргументы: g (главный объект игры), объект PhotoImage (изображение двери, созданное в главе 15), координаты x и y (45 и 30, чтобы дверь располагалась

на платформе вверху игрового экрана), ширину и высоту изображения (40 и 35). Затем добавляем объект door в список спрайтов аналогично остальным спрайтам.

Добежав до двери, человечек остановится не сбоку от нее, а шагнув внутрь:



Код игры целиком

В итоге программа должна состоять из более чем 200 строк кода. Если ваша игра не работает, сверьте каждую функцию (и каждый класс) с этим кодом и исправьте ошибку.

```
from tkinter import *
import random
import time

class Game:
    def __init__(self):
        self.tk = Tk()
        self.tk.title("Человечек спешит к выходу")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                highlightthickness=0)
```

```
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
        self.bg = PhotoImage(file="background.gif")
        w = self.bg.width()
        h = self.bg.height()
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True

    def mainloop(self):
        while 1:
            if self.running == True:
                for sprite in self.sprites:
                    sprite.move()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)

class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def within_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x2):
        return True
    else:
        return False

def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y2):
        return True
    else:
        return False

def collided_left(co1, co2):
    if within_y(co1, co2):
        if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
```

```
            return True
    return False

def collided_right(co1, co2):
    if within_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False

def collided_top(co1, co2):
    if within_x(co1, co2):
        if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
            return True
    return False

def collided_bottom(y, co1, co2):
    if within_x(co1, co2):
        y_calc = co1.y2 + y
        if y_calc >= co2.y1 and y_calc <= co2.y2:
            return True
    return False

class Sprite:
    def __init__(self, game):
        self.game = game
        self.endgame = False
        self.coordinates = None
    def move(self):
        pass
    def coords(self):
        return self.coordinates

class PlatformSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + width, y + height)

class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
        self.images_left = [
            PhotoImage(file="figure-L1.gif"),
            PhotoImage(file="figure-L2.gif"),
            PhotoImage(file="figure-L3.gif")
        ]
        self.images_right = [
            PhotoImage(file="figure-R1.gif"),
            PhotoImage(file="figure-R2.gif"),
```

```
            PhotoImage(file="figure-R3.gif")
        ]
        self.image = game.canvas.create_image(200, 470, \
                image=self.images_left[0], anchor='nw')
        self.x = -2
        self.y = 0
        self.current_image = 0
        self.current_image_add = 1
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

    def turn_left(self, evt):
        if self.y == 0:
            self.x = -2

    def turn_right(self, evt):
        if self.y == 0:
            self.x = 2

    def jump(self, evt):
        if self.y == 0:
            self.y = -4
            self.jump_count = 0

    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
                self.last_time = time.time()
                self.current_image += self.current_image_add
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                       image=self.images_left[self.current_image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                       image=self.images_right[self.current_image])
    def coords(self):
```

```
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates

    def move(self):
        self.animate()
        if self.y < 0:
            self.jump_count += 1
            if self.jump_count > 20:
                self.y = 4
        if self.y > 0:
            self.jump_count -= 1
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
        if self.y > 0 and co.y2 >= self.game.canvas_height:
            self.y = 0
            bottom = False
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            top = False
        if self.x > 0 and co.x2 >= self.game.canvas_width:
            self.x = 0
            right = False
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            left = False
        for sprite in self.game.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
            if bottom and self.y > 0 and collided_bottom(self.y, \
                    co, sprite_co):
                self.y = sprite_co.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                bottom = False
                top = False
            if bottom and falling and self.y == 0 \
                    and co.y2 < self.game.canvas_height \
                    and collided_bottom(1, co, sprite_co):
                falling = False
```

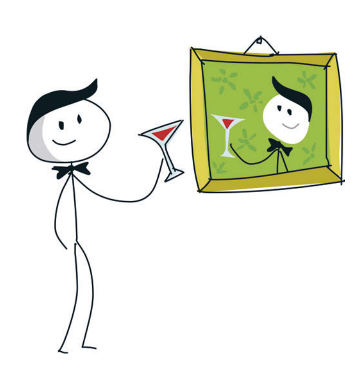
```
                if left and self.x < 0 and collided_left(co, sprite_co):
                    self.x = 0
                    left = False
                    if sprite.endgame:
                        self.game.running = False
                if right and self.x > 0 and collided_right(co, sprite_co):
                    self.x = 0
                    right = False
                    if sprite.endgame:
                        self.game.running = False
            if falling and bottom and self.y == 0 \
                    and co.y2 < self.game.canvas_height:
                self.y = 4
            self.game.canvas.move(self.image, self.x, self.y)

class DoorSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + (width / 2), y + height)
        self.endgame = True
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    0, 480, 100, 10)
platform2 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    150, 440, 100, 10)
platform3 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 400, 100, 10)
platform4 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 160, 100, 10)
platform5 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    45, 60, 66, 10)
platform9 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
```

```
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, PhotoImage(file="door1.gif"), 45, 30, 40, 35)
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```

Что мы узнали

В этой главе мы завершили разработку игры «Человечек спешит к выходу». Создали класс для анимированной фигурки и написали функции, которые позволяют перемещать ее по игровому экрану, чередуя кадры с разными фазами движения. Написали код для определения столкновений фигурки с левой и правой границами холста, а также другими спрайтами, такими как платформы и дверь. Добавили код для определения столкновений с верхней и нижней границами холста и сделали так, чтобы, выбежав за край платформы, фигурка падала вниз. И наконец, добавили проверку, добрался ли человечек до двери (конец игры).



Упражнения

Мы разработали простую игру. Если дополнить программу, игра станет более увлекательной и стильной.

#1. «Вы победили!»

Когда человечек доберется до двери, выводите сообщение «Вы победили!». (Вспомните, как делали то же самое, печатая «Конец игры» для созданной в главе 14 игры «Прыг-скок!».)

#2. Анимация двери

В главе 15 мы создали два изображения двери — открытую и закрытую. Когда человечек в нее войдет, изображение должно поменяться (с закрытой двери на открытую). Затем человечек должен исчезнуть, а дверь закрыться. Тогда возникнет иллюзия, что наш герой открывает дверь, заходит и закрывает ее за собой. Измените для этого код классов `DoorSprite` и `StickFigureSprite`.

#3. Движущиеся платформы

Создайте новый класс для движущихся платформ — `MovingPlatformSprite`. Они должны непрерывно перемещаться по горизонтали (от левой стороны игрового экрана до правой и обратно), из-за чего человечку будет сложнее добраться до выхода.

ПОСЛЕСЛОВИЕ: КУДА ДВИГАТЬСЯ ДАЛЬШЕ

Теперь, когда вы изучили основы программирования и познакомились с языком Python, осваивать другие языки программирования будет гораздо проще. Python — очень мощный язык, но это не значит, что он подойдет для решения любой задачи, поэтому стоит познакомиться и с другими средствами программирования. Сейчас мы рассмотрим несколько альтернативных инструментов для разработки игр и графических приложений, а затем пробежимся по наиболее популярным языкам программирования.

Игры и программирование графики

Если вы хотите серьезно заняться программированием игр и графики, для этого есть немало инструментов. Вот лишь некоторые из них:

- BlitzBasic (*http://www.blitzbasic.com/*) — версия языка BASIC, созданная специально для разработки игр.
- Alice (*http://www.alice.org/*) — среда для программирования 3D-графики.
- Scratch (*http://scratch.mit.edu/*) — визуальный язык программирования, подходящий для создания двухмерных игр.
- Unity3D (*http://unity3d.com/*) — мощная визуальная среда для разработки двухмерных и трехмерных игр.

- Adobe Flash — среда для создания анимаций, работающих в веб-браузере, оснащенная собственным языком программирования ActionScript (*http://www.adobe.com/devnet/actionscript.html*).

В интернете вы найдете множество обучающих ресурсов по каждому из этих инструментов.

Если же вы хотите продолжить работу с Python, вам может пригодиться специально созданная для написания игр библиотека модулей PyGame. Рассмотрим ее подробнее ниже.

PyGame

Чтобы установить PyGame на Windows или Mac OS X, скачайте последнюю версию установочного пакета с веб-страницы *http://www.pygame.org/download.shtml*. А если вы используете Ubuntu, установите PyGame из центра приложений. С документацией к PyGame можно ознакомиться по адресу *http://www.pygame.org/docs/*.

Писать игры с помощью PyGame немного сложнее, чем с `tkinter`. Например, в главе 12 мы использовали `tkinter` для вывода изображения на экран следующим образом:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
myimage = PhotoImage(file='c:\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=myimage)
```

Чтобы сделать то же при помощи PyGame (но загружая *.bmp*-файл, а не *.gif*), понадобится примерно такой код:

```
  import sys
  import time
  import pygame
  import pygame.image as image
  import pygame.display as display
❶ pygame.init()
❷ img = image.load("c:\\test.bmp")
❸ screen = display.set_mode((img.get_width(), img.get_height()))
❹ screen.fill(pygame.Color(255, 255, 255))
❺ screen.blit(img, (0, 0))
❻ display.flip()
❼ time.sleep(10)
❽ display.quit()
```

После импортирования модулей вызываем функцию `init` модуля `pygame` в строке ❶. Это приблизительно соответствует созданию холста и вызову функции `pack` в примере для `tkinter`. В строке ❷ загружаем *.bmp*-файл с изображением, который мы скопировали или создали предварительно в той же папке, где размещается наша программа, с помощью функции `load` PyGame-модуля `image`. В строке ❸ создаем объект `screen`, вызывая функцию `set_mode` PyGame-модуля `display`. В строке ❹ (без которой можно и обойтись) очищаем экран, заполняя его белым цветом. В строке ❺ вызываем функцию объекта-экрана `blit`, которая выводит на экран изображение. Эта функция принимает само изображение, а также кортеж с позицией, где мы хотим его вывести (отступ 0 пикселей слева и 0 пикселей сверху).

Load — загрузить

Screen — экран

Set mode — задать режим

Display — показать, экран

PyGame использует *внеэкранный буфер* (также это называют *двойной буферизацией*), то есть формирует экранную картинку в памяти компьютера, где она невидима, и переносит ее на экран. Двойная буферизация уменьшает эффект мерцания, который может возникнуть при выводе движущихся объектов на экран. В нашем примере копирование внеэкранного буфера на экран выполняет функция `flip` в строке ❻.

В строке ❼ делаем 10-секундную паузу, поскольку, в отличие от холста `tkinter`, окно PyGame при завершении программы сразу же закрывается. В строке ❽ вызываем функцию `display.quit`, чтобы PyGame корректно завершила работу. PyGame, конечно, способна на большее, но по этому примеру уже можно составить о ней первое впечатление.

Quit — выйти

Языки программирования

Среди наиболее популярных языков программирования кроме Python можно отметить Java, C/C++, C#, PHP, Objective-C, Perl, Ruby и JavaScript. Рассмотрим эти языки, включая код программы «Привет, мир» (аналог Python-версии из главы 1) для каждого из них. Имейте в виду, что перечисленные языки заметно отличаются от Python и не ориентированы на начинающих программистов. Поскольку не все языки позволяют выводить русский текст так же просто, как Python, будем печатать английское «Hello World» вместо «Привет, мир», чтобы не усложнять код примеров.

Java

Java (*http://www.oracle.com/technetwork/java/index.html*) — это язык программирования средней сложности, укомплектованный обширной библиотекой модулей, которые называются пакетами. В интернете много бесплатной документации по Java. Использовать этот язык можно практически во всех операционных системах. Java — основной язык, на котором создаются приложения для мобильных телефонов под управлением Android.

Программа «Привет, мир» на языке Java выглядит так:

```
public class HelloWorld {
    public static final void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Hello world — привет, мир

C/C++

C (*http://www.cprogramming.com/*) и C++ (*http://www.stroustrup/C++.html*) — это сложные языки программирования, доступные для всех операционных систем, причем есть как бесплатные, так и коммерческие версии. В частности, многие действия, которые в Python выполняются автоматически, в C/C++ придется программировать самостоятельно (например, запрашивать у компьютера блок памяти для хранения объекта). Многие коммерческие игры, в том числе для игровых приставок, написаны на том или ином диалекте C или C++.

Вот программа «Привет, мир» на языке C:

```
#include <stdio.h>
int main ()
{
  printf ("Hello World\n");
}
```

А вот «Привет, мир» на C++:

```
#include <iostream>
int main()
{
  std::cout << "Hello World\n";
}
```

C#

C# (*http://msdn.microsoft.com/en-us/vstudio/hh388566/*) — произносится «си шарп» — язык средней сложности, предназначенный в основном для Windows-систем и очень похожий на Java. Программировать в C# проще, чем в C и C++.

Вот пример программы «Привет, мир» на C#:

```
public class Hello
{
   public static void Main()
   {
      System.Console.WriteLine("Hello World");
   }
}
```

PHP

PHP (*http://www.php.net/*) — язык программирования для создания сайтов. Для разработки на PHP нужен сервер (программное обеспечение, отсылающее веб-страницы браузеру) с установленным PHP. Необходимые программы можно скачать бесплатно для всех популярных операционных систем. Чтобы создавать сайты на PHP, понадобится изучить HTML (простой язык разметки для создания веб-страниц). По адресу *http://php.net/manual/ru/tutorial.php* можно ознакомиться с обучающими статьями по PHP, а по адресу *http://htmlbook.ru/samhtml* — с самоучителем по HTML.

HTML-страница, выводящая фразу «Hello World» в окне браузера, будет примерно такой:

```
<html>
    <body>
        <p>Hello World</p>
    </body>
</html>
```

А вот PHP-страница, выводящая это же сообщение:

```
<?php
echo "Hello World\n";
?>
```

Objective-C

Язык Objective-C (*http://www.tutorialspoint.com/objective_c/*) очень похож на C (фактически, это расширение языка C). Используют его в основном на компьютерах фирмы Apple. На Objective-C написано большинство приложений для iPhone и iPad.

Программа «Привет, мир» на Objective-C:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSLog (@"Hello World");

   [pool drain];
   return 0;
}
```

Perl

Язык программирования Perl (*http://www.perl.org/*) доступен бесплатно для всех основных операционных систем. Обычно его используют для создания сайтов (аналогично PHP).

Вот программа «Привет, мир» на Perl:

```
print("Hello World\n");
```

Ruby

Ruby (*http://www.ruby-lang.org/*) — язык программирования, доступный для всех популярных операционных систем бесплатно. В основном он используется для создания сайтов совместно с фреймворком Ruby on Rails (*фреймворком* называют набор программных библиотек для облегчения разработки приложений определенного типа).

Программа «Привет, мир» на Ruby:

```
puts "Hello World"
```

JavaScript

JavaScript (*https://developer.mozilla.org/ru/docs/Web/JavaScript*) — язык, который изначально предназначался для встраивания в веб-страницы, однако все чаще его используют при создании видеоигр. Синтаксисом JavaScript напоминает Java, но в освоении он проще (вы можете создать HTML-страницу со встроенной JavaScript-программой и запустить ее в браузере без использования оболочки, командной строки и других инструментов). Изучение JavaScript можно начать с курсов на сайте Codecademy (*http://www.codecademy.com/)*.

Программа «Привет, мир» на JavaScript может выглядеть по-разному, в зависимости от того, вводить код в консоли браузера или внедрять его в HTML-страничку. В консоли код будет таким:

```
print('Hello World');
```

А внутри HTML-страницы — таким:

```
<html>
    <body>
        <script type="text/javascript">
            alert("Hello World");
        </script>
    </body>
</html>
```

Заключение

Остановитесь ли вы на Python или выберете другой язык программирования (а их гораздо больше, чем я перечислил) — почерпнутые из этой книги знания вам обязательно пригодятся. Даже если вы не будете заниматься программированием, понимание его основ поможет в разных ситуациях, как в школе, так и потом на работе.

Удачи вам и увлекательного программирования!

ПРИЛОЖЕНИЕ: КЛЮЧЕВЫЕ СЛОВА PYTHON

В Python и большинстве других языков программирования *ключевыми* называют слова, которые являются частью самого языка, и поэтому их нельзя больше ни для чего использовать. Например, при попытке назвать ключевым словом переменную вместо работающей программы вы получите сообщения об ошибках — порой забавные, а порой загадочные.

В этом приложении описаны все ключевые слова языка Python. Используйте это приложение как справочный материал при дальнейшем изучении программирования.

and

And — и

Ключевым словом and обозначается операция «логическое И». Ее используют для объединения двух булевых (то есть возвращающих True или False) выражений. Получившееся составное выражение даст True, только если оба исходных выражения возвращают True. Пример:

```
if age > 10 and age < 20:
    print('Осторожно, подросток!!!!')
```

Age — возраст
Print — напечатать

Сообщение будет напечатано лишь в том случае, если значение переменной age больше 10 *и* меньше 20.

As — как

as

С помощью ключевого слова as можно дать импортированному модулю другое имя. Допустим, у нас есть модуль с очень длинным названием:

I am a Python module that is not very useful — я модуль Python, который не очень полезен

```
i_am_a_python_module_that_is_not_very_useful
```

Вводить такое имя при каждом обращении к модулю будет неудобно:

Do something — сделать что-то

Do something else — сделать что-то другое

```
import i_am_a_python_module_that_is_not_very_useful
i_am_a_python_module_that_is_not_very_useful.do_something()
Я сделал что-то бесполезное.
i_am_a_python_module_that_is_not_very_useful.do_something_else()
Я сделал еще что-то бесполезное!
```

При импортировании модулю лучше дать новое короткое имя (псевдоним):

Not useful — не полезный

```
import i_am_a_python_module_that_is_not_very_useful as notuseful
notuseful.do_something()
Я сделал что-то бесполезное.
notuseful.do_something_else()
Я сделал еще что-то бесполезное!
```

Assert — утверждать

assert

Ключевое слово assert позволяет убедиться, что некое выражение дает True. Это один из способов обнаружить ошибки и неполадки. Он применяется при создании более сложных программ, поэтому в коде примеров в этой книге assert не используется. Вот простая конструкция assert:

My number — мое число

```
>>> mynumber = 10
>>> assert mynumber < 5
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert a < 5
AssertionError
```

Assertion Error — ошибка утверждения

В этом примере мы хотим убедиться, что значение переменной mynumber меньше 5. Поскольку в данном случае это не так, Python останавливает программу и выводит ошибку AssertionError.

break

Ключевое слово `break` служит для досрочного выхода из цикла. Например, для цикла `for`:

Break — остановить

For — для

```
age = 10
for x in range(1, 100):
    print('отсчет %s' % x)
    if x == age:
        print('закончили считать')
        break
```

Поскольку в переменной `age` хранится число 10, эта программа напечатает вот что:

```
отсчет 1
отсчет 2
отсчет 3
отсчет 4
отсчет 5
отсчет 6
отсчет 7
отсчет 8
отсчет 9
отсчет 10
закончили считать
```

Как только значение переменной `x` достигнет 10, программа выведет сообщение «Закончили считать» и выйдет из цикла.

class

Class — класс

Ключевое слово `class` служит для определения типов сущностей (классов), таких как «средство передвижения», «животное», «человек». У каждого класса может быть специальная функция `__init__`, предназначенная для подготовки новых объектов к работе. Например, если объектам класса `Car` нужно свойство `color`, его можно добавить в функции `__init__`, которая вызывается при создании объекта:

Init — от initiate — запустить, начать

Car — машина

Color — цвет

```
class Car:
    def __init__(self, color):
        self.color = color
```

```
car1 = Car('red')
car2 = Car('blue')
print(car1.color)
red
print(car2.color)
blue
```

Red — красный

Blue — синий

continue

Continue — продолжить

Ключевое слово continue позволяет «перескочить» к следующему повтору цикла, так что следующая после continue часть тела цикла будет пропущена. В отличие от break происходит не выход из цикла, а досрочное завершение текущего повтора с переходом к следующему. Например, если у нас есть список строк и мы хотим напечатать все элементы, которые начинаются не с буквы «б», можно написать такой код:

My items — мои предметы

Starts with — начинается с

```
❶ >>> my_items = ['яблоко', 'трубкозуб', 'банан', 'барсук',
                  'клементин', 'верблюд']
❷ >>> for item in my_items:
❸         if item.startswith('б'):
❹             continue
❺         print(item)

    яблоко
    трубкозуб
    клементин
    верблюд
```

В строке ❶ создаем список строк, в строке ❷ задаем цикл для перебора элементов списка. В строке ❸ выполняется проверка, начинается ли текущий элемент с буквы «б». Если это так, в строке ❹ с помощью continue переходим к следующему повтору цикла (и следующему элементу списка). Если нет, печатаем текущий элемент в строке ❺.

def

Def — от define — определить

Используйте ключевое слово def, чтобы дать определение функции. Например, так можно определить функцию, преобразующую годы в минуты:

Minutes — минуты

Years — годы

```
>>> def minutes(years):
        return years * 365 * 24 * 60
>>> minutes(10)
5256000
```

del

Del — от delete — удалить

С помощью ключевого слова `del` можно удалить элемент из списка или словаря. Предположим, вы от руки составили список подарков, которые хотите получить на день рождения, затем вычеркнули из списка одну строку и добавили новую:

радиоуправляемый автомобиль
новый велосипед
~~*компьютерная игра*~~
змея-робот

В Python изначальный список будет выглядеть так:

```
what_i_want = ['радиоуправляемый автомобиль', 'новый велосипед',
'компьютерная игра']
```

What I want — что я хочу

Чтобы удалить из списка компьютерную игру, используем команду `del` с индексом соответствующего элемента. Затем можно добавить новый элемент с помощью функции `append`:

Append —добавить

```
del what_i_want[2]
what_i_want.append('змея-робот')
```

И вывести измененный список на экран:

```
print(what_i_want)
['радиоуправляемый автомобиль', 'новый велосипед', 'змея-робот']
```

elif

Ключевое слово `elif` используется как часть конструкции `if` (см. описание ключевого слова `if`).

else

Else — еще

Ключевое слово `else` используется как часть конструкции `if` (см. описание ключевого слова `if`).

Except — *здесь* от exception — исключение

except

Ключевое слово except служит для обнаружения неполадок в коде. Как правило, его используют в сложных программах, поэтому в нашей книге мы его не рассматриваем.

Finally — под конец

finally

Ключевое слово finally нужно для выполнения некоего обязательного кода при возникновении ошибки (как правило, чтобы освободить ресурсы, затребованные программой перед тем, как возникла ошибка). В этой книге ключевое слово finally не рассматривается, поскольку относится к продвинутым приемам программирования.

For — для

for

Ключевое слово for используется для создания цикла, повторяющего блок кода определенное число раз. Например:

```
for x in range(0, 5):
    print('x = %s' % x)
```

Этот цикл повторит блок кода (команду print) пять раз, после чего вы увидите на экране:

```
x = 0
x = 1
x = 2
x = 3
x = 4
```

From — из

from

Turtle — черепаха

Pen — ручка

Ключевое слово from позволяет импортировать только ту часть модуля, которая необходима. Например, в модуле turtle (впервые упоминается в главе 4) есть класс Pen для создания холста, по которому перемещается черепашка. Можно импортировать весь модуль turtle, затем использовать класс Pen:

```
import turtle
t = turtle.Pen()
```

А можно импортировать только класс Pen и затем использовать его без явного обращения к модулю turtle:

```
from turtle import Pen
t = Pen()
```

Один из плюсов данного подхода в том, что, заглянув в начало программы, вы сразу увидите, какие функции и классы в ней задействованы (это особенно актуально для программ, импортирующих множество модулей). Однако вы не сможете использовать те части модулей, которые не были импортированы. Например, в модуле time есть функции localtime и gmtime. Если вы импортируете только localtime, а потом попытаетесь воспользоваться gmtime, возникнет ошибка:

```
>>> from time import localtime
>>> print(localtime())
(2007, 1, 30, 20, 53, 42, 1, 30, 0)
>>> print(gmtime())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'gmtime' is not defined
```

Time — время

Local time — местное время

GM time — от Greenwich Mean — время по Гринвичу

Сообщение об ошибке — «name 'gmtime' is not defined» — означает, что Python ничего не знает о функции gmtime. И это неудивительно, поскольку вы ее не импортировали.

Если в каком-то модуле содержится набор нужных вам функций, но вы не хотите при каждом обращении к ним указывать имя модуля (например, time.localtime или time.gmtime), вы можете импортировать все содержимое модуля с помощью знака звездочки (*). Вот так:

```
>>> from time import *
>>> print(localtime())
(2007, 1, 30, 20, 57, 7, 1, 30, 0)
>>> print(gmtime())
(2007, 1, 30, 13, 57, 9, 1, 30, 0)
```

Импортируется все содержимое модуля time, и можно использовать входящие в него функции, указывая только их имена.

global

Global — глобальный

Концепция области видимости рассматривается в главе 7. Определенная вне функции переменная, как правило, доступна (видима) в коде этой

функции, но определенная внутри функции переменная снаружи обычно не видна. Ключевое слово `global` позволяет сделать исключение из этого правила: переменная, объявленная как глобальная (`global`), видима во всей программе. Например:

```
>>> def test():
        global a
        a = 1
        b = 2
```

Если вызвать функцию `test`, затем дать команды `print(a)` и `print(b)`, первая команда сработает, а вторая нет (произойдет ошибка):

```
>>> test()
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Переменная `a` была объявлена в функции `test` как глобальная, поэтому видима даже вне этой функции. Однако `b` видима только в коде функции, и чтобы обращаться к ней так же, как к `a`, ее тоже придется сделать глобальной.

If — если

if

Ключевое слово `if` нужно для выполнения кода в соответствии с заданным условием. Совместно с `if` также используются ключевые слова `else` и `elif` (сокращение от else if). Например:

Toy price — цена игрушки

```
❶ if toy_price > 1000:
❷     print('Эта игрушка чересчур дорогая')
❸ elif toy_price > 100:
❹     print('Эта дорогая игрушка')
❺ else:
❻     print('Я могу купить эту игрушку')
```

Если, согласно условию в строке ❶, хранящаяся в переменной `toy_price` цена игрушки превышает 1000, в строке ❷ будет напечатано сообщение о ее чрезмерной дороговизне. В противном случае будет выполнена проверка в строке ❸: если цена игрушки больше 100, в строке ❹ будет напечатано сообщение, что игрушка дорогая. И наконец, если

ни одно из предыдущих условий не выполняется, сработает ветка else в строке ❺, в результате чего в строке ❻ будет напечатано сообщение «Я могу купить эту игрушку».

import

Import — импортировать

Ключевое слово import используется для загрузки (импортирования) модуля в программу. Например, следующая команда загружает модуль sys, после чего им можно пользоваться:

```
import sys
```

Sys — от system — система

in

In — в

Ключевое слово in используется в логических выражениях для проверки, входит ли некое значение в набор элементов. Например, есть ли среди элементов списка число 1:

```
>>> if 1 in [1,2,3,4]:
>>>     print('число есть в списке')
число есть в списке
```

А так можно проверить, находится ли строка «штаны» в списке предметов одежды:

```
>>> clothing_list = ['шорты', 'трусики', 'боксеры', 'кальсоны',
'панталоны']
>>> if 'штаны' in clothing_list:
        print('штаны есть в списке')
else:
        print('штанов в списке нет')
штанов в списке нет
```

Clothing list — список вещей

is

Is — есть

Ключевое слово is по смыслу похоже на оператор «равно» (==), который используется для проверки равенства двух значений (например, выражение 10 == 10 даст True, а 10 == 11 даст False). Однако между is и == есть различия. Если сравнение значений через == дает True, сравнение тех же значений через is в некоторых случаях может дать False (даже если значения одинаковы). Это относится к продвинутым средствам программирования, поэтому в книге мы всегда используем оператор ==.

Lambda — лямбда, в прогр. анонимная функция

lambda

Ключевое слово lambda используется для создания безымянных (или «встроенных») функций. Мы не рассматриваем безымянные функции в этой книге.

Not — не

not

Если утверждение истинно, ключевое слово not сделает его ложным. Например, если создать переменную x и задать ей значение True, а затем напечатать значение x, поставив перед ним not, выйдет вот что:

```
>>> x = True
>>> print(not x)
False
```

Ключевое слово not кажется бесполезным до тех пор, пока вы не начнете использовать not вместе с if. Например, проверить, что строка *не* находится в списке, можно так:

```
>>> clothing_list = ['шорты', 'трусики', 'боксеры', 'кальсоны',
'панталоны']
>>> if 'штаны' not in clothing_list:
        print('Как вы живете без штанов?')
Как вы живете без штанов?
```

Or — или

or

Ключевым словом or обозначается операция «логическое ИЛИ». Ее используют для объединения двух булевых (то есть возвращающих True или False) выражений. Составное выражение даст True, если хотя бы одно из исходных выражений дает True. Например:

Dino — от dinosaur — динозавр

```
if dino == 'Тираннозавр' or dino == 'Аллозавр':
    print('Хищники')
elif dino == 'Анкилозавр' or dino == 'Апатозавр':
    print('Травоядные')
```

Если переменная dino содержит строку «Тираннозавр» *или* «Аллозавр», программа напечатает: «Хищники». Если же dino содержит строку «Анкилозавр» *или* «Апатозавр», программа напечатает: «Травоядные».

pass

Pass — пропустить

Иногда программу удобнее писать и тестировать небольшими частями. Но есть проблема: нельзя создать конструкцию if без блока с кодом, который должен выполняться, когда условие if дает True. Точно так же не получится создать цикл for без тела цикла. Например, этот код работает нормально:

```
>>> age = 15
>>> if age > 10:
        print('старше 10')

старше 10
```

Однако если убрать тело if, Python выдаст ошибку:

```
>>> age = 15
>>> if age > 10:

File "<stdin>", line 2
^
IndentationError: expected an indented block
```

Такое сообщение об ошибке возникает, если после какой-либо конструкции должен стоять блок кода, но его там нет (кстати, IDLE не позволит ввести код этого примера). В подобных случаях вместо отсутствующего кода можно ввести команду pass, которая не выполняет действий.

Допустим, мы хотим создать цикл for с конструкцией if внутри него, но пока не решили, что будет происходить в теле if (печать сообщения, выход из цикла по break или еще что-то). Можно воспользоваться pass, и код запустится, несмотря на то что он не завершен.

Вот конструкция if с ключевым словом pass в качестве блока кода:

```
>>> age = 15
>>> if age > 10:
        pass
```

Еще пример использования pass:

```
>>> for x in range(0, 7):
>>>     print('x = %s' % x)
```

```
>>>     if x == 5:
            pass

x = 0
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
```

Python при каждом повторе цикла проверяет, содержит ли переменная x число 5, но если это так, не выполняет действий, а просто печатает числа в диапазоне от 0 до 7.

Можно заменить pass работающим кодом, например командой break:

```
>>> for x in range(1, 7):
        print('x = %s' % x)
        if x == 5:
            break

x = 1
x = 2
x = 3
x = 4
x = 5
```

Но чаще всего ключевое слово pass используют для определения функции без написания кода, чтобы добавить его позже.

Raise — поднять

raise

С помощью ключевого слова raise можно вызвать ошибку в программе. Как ни странно, в продвинутом программировании это часто бывает полезным. В этой книге команда raise не рассматривается.

Return — вернуть

return

Ключевое слово return используется для возвращения значения из функции. Например, можно создать функцию, которая вычисляет количество секунд, прожитых вами до последнего дня рождения:

Age in seconds — возраст в секундах

Age in years — возраст в годах

```
def age_in_seconds(age_in_years):
    return age_in_years * 365 * 24 * 60 * 60
```

Эта функция возвращает значение, которое можно присвоить переменной или вывести на экран:

```
>>> seconds = age_in_seconds(9)
>>> print(seconds)
283824000
>>> print(age_in_seconds())
378432000
```

try

Try — попробовать

Ключевое слово try открывает блок кода, который заканчивается ключевыми словами except и finally. Блоки try/except/finally используются для обработки ошибок (например, чтобы показывать пользователю понятное сообщение, а не ошибку Python). В этой книге ключевое слово try не рассматривается.

while

While — пока

Цикл while напоминает for, однако если for перебирает значения из заданного диапазона, то while повторяется до тех пор, пока условие дает True. Будьте осторожны: если условие цикла while всегда будет выполняться, программа не выйдет из цикла (это называется бесконечным циклом), пока вы ее не остановите, нажав Ctrl-C. Например:

```
>>> x = 1
>>> while x == 1:
        print('Привет')
```

Следующий код напечатает «Привет» девять раз (на каждом повторе увеличивая x на 1, пока x не станет равен 10):

```
>>> x = 1
>>> while x < 10:
        print('Привет')
        x = x + 1
```

with

With — с

Ключевое слово with используется для создания блоков кода с обработкой ошибок наподобие блоков try/finally. В этой книге команда with не рассматривается.

Yield — здесь «создавать»

yield

Ключевое слово `yield` по смыслу напоминает `return`, но используется со специальным классом объектов — генераторами. Генераторы создают значения «на лету» (то есть по запросу), и функция `range` работает как генератор. В этой книге команда `yield` не рассматривается.

ГЛОССАРИЙ

Если вы только начали знакомиться с программированием, время от времени вам будут встречаться непонятные термины и это может помешать в освоении нового материала.

В глоссарии вы найдете определения многих понятий из области программирования, так что, встретив на страницах этой книги непонятное слово, заглядывайте сюда.

Null — отсутствие значения (в Python вместо `null` используется ключевое слово `None`).

Анимация — поочередный вывод на экран набора изображений с большой скоростью, чтобы возникла иллюзия движения.

Аргумент — значение, передаваемое при вызове функции или создании объекта (в Python при создании объекта вызывается функция `__init__`).

Блок — сгруппированная последовательность команд в программе.

Булев тип — тип данных, предполагающий только два возможных значения — «Истина» и «Ложь». В Python «Истина» обозначается ключевым словом `True`, а «Ложь» — `False` (с прописных букв T и F).

Вертикальный — направление сверху вниз на экране (координата *y*).

Встраивание значений — вставка значений внутрь строки (обычно в позициях, обозначенных специальными метками).

Вызов — выполнение кода функции. Используя функцию, мы говорим, что вызываем ее.

Выполнение — запуск кода (программы, фрагмента программы, функции).

Горизонтальный — направление слева направо на экране (соответствует координате *x*).

Градус — единица измерения углов.

Данные — информация, которая обрабатывается компьютером.

Диалог (диалоговое окно) — небольшое окно в программе для выполнения неких действий: вывода предупреждения, сообщения об ошибке, запроса дополнительных данных и т. п. Например, при выборе пункта меню «Открыть файл» программа выводит диалог открытия файла.

Идентификатор — уникальное число, обозначающее конкретный объект в программе. Например, в модуле `tkinter` идентификаторы используются для обозначения фигур на экране.

Измерения — в компьютерной графике изображения могут быть в двух (2D-графика) и трех (3D-графика) измерениях. Двухмерная (2D) графика — это плоские экранные изображения (как картинки в старых мультфильмах), которые характеризуются шириной и высотой. К трехмерной (3D) графике относятся изображения, обладающие шириной, высотой и глубиной. Такая графика часто встречается в современных компьютерных играх.

Изображение — картинка на экране компьютера.

Импортирование — операция, которая делает модуль доступным для использования в программе.

Инициализация — установка начального состояния объекта (свойств объекта при его создании).

Инсталляция (установка) — процесс копирования файлов приложения на компьютер, после чего приложение становится доступным для использования.

Исключение — разновидность ошибки, которая может произойти во время выполнения программы.

Кадр — одно из изображений, составляющих анимацию.

Класс — описание или определение типа сущностей. В программировании класс представляет собой набор функций и свойств (переменных), которые присущи всем объектам данного класса.

Клик — нажатие одной из кнопок мышки для совершения действия на экране (выбора опции меню и т. п.).

Ключевое слово — специальное слово, используемое в языке программирования. Ключевые слова также называют зарезервированными. Это значит, что их нельзя использовать произвольно (например, в качестве имен переменных).

Координаты — позиция точки на экране. Обычно состоит из отступов по горизонтали (*x*-координата) и вертикали (*y*-координата).

Модуль — набор функций и переменных.

Область видимости переменной — часть программы, где эту переменную можно использовать. Например, переменная, созданная внутри функции, будет видима в теле функции и недоступна (невидима) за ее пределами.

Оболочка — интерфейс командной строки. Упоминаемая в данной книге оболочка Python — это командный интерфейс, встроенный в программу IDLE.

Объект — конкретный экземпляр класса. При создании каждого объекта программа выделяет область памяти для хранения принадлежащих ему данных.

Оператор — элемент компьютерной программы, используемый для математических вычислений или сравнения значений.

Ошибка — ситуация, когда компьютерная программа работает неправильно. При возникновении ошибки Python выводит сообщение. Например, введя код с неверными отступами, вы получите сообщение об ошибке `IndentationError`.

Память — устройство в компьютере, используемое для хранения информации.

Папка (директория, каталог) — местонахождение группы файлов на диске компьютера.

Переменная — способ хранения значений в программе. Переменная похожа на ярлык с названием, которым помечают фрагмент данных, хранящийся в памяти компьютера. Значения переменных могут меняться, поэтому они так и называются.

Пиксель — самая маленькая точка на экране, которую компьютер способен отобразить.

Потомок — между классами в программе могут существовать отношения «предок — потомок» (это называется наследованием). Класс-потомок наследует характеристики класса-предка.

Предок класса — другой класс, от которого первый класс наследует функции и свойства. Класс-потомок наследует характеристики класса-предка.

Программа — набор команд, объясняющий компьютеру, что и как ему делать.

Программное обеспечение (софт) — набор компьютерных программ.

Прозрачность — часть изображения, которая не выводится на экран и поэтому не заслоняет графические объекты заднего плана.

Синтаксис — набор правил по составлению текстов программ на языке программирования.

Событие — нечто, происходящее во время выполнения программы. Например: событие перемещения мышки, нажатия ее кнопки или ввода с клавиатуры.

Спрайт — персонаж или иной графический объект в компьютерной игре.

Столкновение — в компьютерных играх ситуация, когда один персонаж игры сталкивается с другим персонажем или каким-нибудь экранным объектом.

Строка — набор алфавитно-цифровых символов (букв, чисел, знаков препинания и пробелов).

Условие — выражение в программе, представляющее собой вопрос, на который возможны два ответа: «Истина» (`True`) либо «Ложь» (`False`).

Функция — команда в программе, которая запускает последовательность других команд, выполняющих некое действие.

Холст — область экрана, на которой можно рисовать. В модуле `tkinter` холсту соответствует класс `Canvas`.

Цикл — многократное выполнение команды или набора команд.

Шестнадцатеричная система — широко используемый в программировании способ представления чисел. В шестнадцатеричной системе используется основание 16, из-за чего применяется не 10, а 16 цифр: обычные — от 0 до 9, и буквенные — A, B, C, D, E и F.

Экземпляр класса — объект, принадлежащий этому классу.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Л

М

Н

О

С

Т

Ч

Ш

Э

Я

ОБ АВТОРЕ

Джейсон Р. Бриггс начал заниматься программированием в восемь лет, едва изучив BASIC на компьютере Radio Shack TRS-80. Разработчик и системный архитектор, кроме того Джейсон был редактором журнала Java Developer's Journal. Его статьи публиковались в изданиях JavaWorld, ONJava и ONLamp. «Python для детей» — первая книга Джейсона.

С Джейсоном можно связаться через его сайт *http://jasonrbriggs.com/* или по электронной почте *mail@jasonrbriggs.com*.

О ХУДОЖНИКЕ

Миран Липовача — автор книги «Изучай Haskell во имя добра!». Обожает боксировать, играет на бас-гитаре, рисует и любит фантазировать. Он неравнодушен к танцующим скелетам и числу 71, а проходя через автоматические двери, делает вид, будто открывает их силой разума.

О ТЕХНИЧЕСКИХ РЕЦЕНЗЕНТАХ

Недавний выпускник школы 15-летний Джош Поллок учится на первом курсе высшей школы Lick-Wilmerding в Сан-Франциско. В девять лет начал программировать на языке Scratch, в шестом классе изучил TI-BASIC, в седьмом стал осваивать Java и Python, а в восьмом занялся UnityScript. Помимо программирования любит играть на трубе, разрабатывать компьютерные игры и обучать людей интересным концепциям в области точных наук.

Мария Фернандес — магистр по прикладной лингвистике, увлекается компьютерами и техникой более 20 лет. Преподавала английский язык девушкам-беженцам в рамках инициативы Global Village Project в Джорджии. Живет в Северной Калифорнии, работает в ETS (некоммерческая организация, занимающаяся проверкой и оценкой систем образования).

БЛАГОДАРНОСТИ

Это как подняться на сцену за наградой и обнаружить, что список людей, которым нужно выразить благодарность, остался в других брюках: ты наверняка забудешь про кого-то и вот-вот грянет музыка, означающая, что пора закругляться.

Тем не менее ниже я привожу, без сомнения, неполный список людей, которым я бесконечно признателен за то, что они помогли мне сделать эту книгу такой, какая она сейчас! А на мой взгляд, она очень хороша!

Спасибо команде No Starch Press, и в особенности Биллу Поллоку, за редактирование книги с поправками на восприятие детей школьного возраста. Когда ты давно занимаешься программированием, забываешь, насколько сложны некоторые моменты для начинающих, и помощь Билла в их выявлении была неоценимой. Спасибо Сирене Янг — выдающемуся директору производства, которая очень постаралась, чтобы 300 с лишним страниц кода были должным образом раскрашены.

Огромное спасибо Мирану Липоваче за превосходные иллюстрации! Если бы ими занимался я, получились бы некие смутные контуры, непохожие ни на что конкретное. Это медведь или собака? Нет, видимо, это дерево!

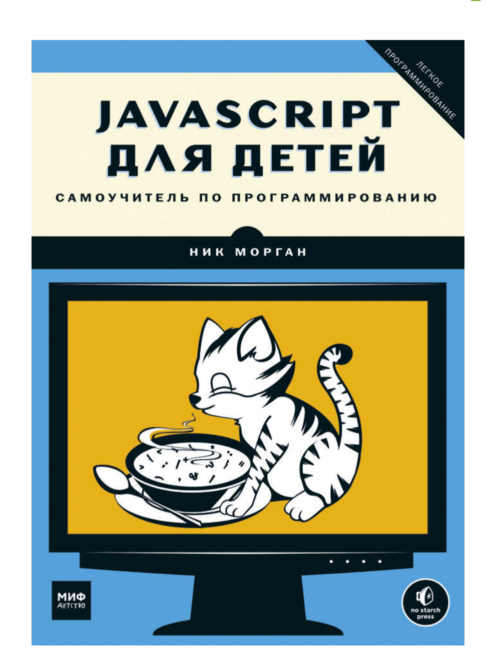
Спасибо рецензентам. Простите, если некоторые из ваших предложений так и не были воплощены в окончательном варианте книги. Полагаю, вы были правы, и в возможных оплошностях стоит винить лишь мой непростой характер. Отдельное спасибо Джошу за ряд ценных идей и замечательных находок. Приношу свои извинения Марии, за то что ей пришлось повозиться с кодом, который местами был плохо отформатирован.

Спасибо моей жене и дочке за понимание. Ведь их муж и отец еще чаще, чем обычно, сидел, уткнувшись в экран компьютера.

Спасибо маме за безграничную поддержку на протяжении многих лет.

И наконец, спасибо моему отцу за покупку компьютера в далеком 1970 году и терпеливое отношение к ребенку, который хотел пользоваться компьютером так же часто, как он. Без него эта книга не появилась бы!

УЧИТЕСЬ И НА ДРУГИХ ЯЗЫКАХ



Javascript для детей. Самоучитель по программированию

Ник Морган

Эта книга позволит вам погрузиться в программирование и с легкостью освоить JavaScipt. Вы напишете несколько настоящих игр — поиск сокровищ на карте, «Виселицу» и «Змейку». На каждом шаге вы сможете оценить результаты своих трудов — в виде работающей программы, а с понятными инструкциями, примерами и забавными иллюстрациями обучение будет только приятным. Книга для детей от 10 лет.



Scratch для детей. Самоучитель по программированию

Мажед Маржи

Scratch — простой, понятный и невероятно веселый язык программирования для детей. В нем нет кода, который нужно знать назубок и писать без ошибок. Все, что требуется, — это умение читать и считать. Как из конструктора, при помощи Scratch можно собирать программы из разноцветных «кирпичиков» — блоков. В программу можно вносить любые изменения в любой момент и сразу видеть, как она работает.

Подробные объяснения, разобранные по шагам примеры и множество упражнений помогут освоить Scratch без труда. Эта книга подойдет детям от 8 лет (и их родителям!), а также всем, кто хочет научиться программировать с нуля.

ПРОГРАММИРОВАТЬ С КНИГАМИ СЕРИИ!

Программируем с Minecraft. Создай свой мир с помощью Python

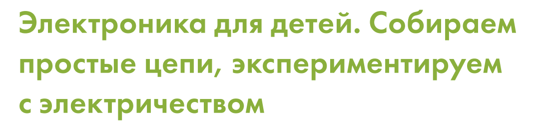
Крэйг Ричардсон



Ты совладал с криперами, спускался в глубочайшие пещеры и, быть может, даже добрался до Края и вернулся обратно, но доводилось ли тебе сделать из меча волшебную палочку? Или построить дворец в мгновение ока? Или обустроить свой личный, меняющий цвет танцпол?

Эта книга покажет, как сотворить эти и многие другие чудеса, используя силу Python, бесплатного языка программирования, которым пользуются миллионы программистов — и новички, и профи!

Начни изучение Python с кратких, несложных уроков и используй эти навыки для преображения мира Minecraft, получая мгновенные потрясающие результаты! Узнай, как настроить Minecraft под себя, создавая мини-игры, клонируя целые здания и превращая обычные скучные блоки в золото.



Электроника для детей. Собираем простые цепи, экспериментируем с электричеством

Эйвинд Нюдаль Даль



Если вы когда-нибудь смотрели на электронное устройство и задавались вопросом «Как оно работает?» или «Могу ли я сделать это сам?», то вы нашли то, что нужно. И не важно, 8 вам лет или 100, вас ждут увлекательные эксперименты, захватывающие задания и грандиозный проект в конце: нужно будет создать игру, в которую вы сможете играть с друзьями.

Издание для досуга
Для широкого круга читателей

Бриггс Джейсон

Python для детей
Самоучитель по программированию

Главный редактор *Артем Степанов*
Руководитель направления *Анастасия Троян*
Продюсер проекта *Евгения Рыкалова*
Ответственный редактор *Анна Дружинец*
Литературный редактор *Елена Ефремова*
Научный редактор *Дарья Абрамова*
Дизайн обложки *Сергей Хозин*
Верстка *Елена Бреге*
Корректоры *Елена Пинчукова, Надежда Болотина*

НАУЧИТ ПРОГРАММИРОВАТЬ С НУЛЯ



ОДИН ИЗ САМЫХ ВОСТРЕБОВАННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Python — язык, на котором можно запрограммировать любой алгоритм. Самоучитель погрузит вас в мир настоящего программирования и расскажет об основах работы с одним их самых востребованных языков. Свои знания вы сможете проверить сразу же — на забавных примерах и уморительно смешных заданиях, справиться с которыми помогут прожорливые монстры, секретные агенты и воришки-вороны.

Книга подойдет детям от 10 лет (и их родителям!), а также всем, кто хочет начать программировать с нуля на любом языке или освоить именно Python.

Вы узнаете, как:
— использовать основные элементы Python — списки, функции, модули;
— рисовать при помощи встроенных инструментов Python;
— анимировать изображения с tkinter;
— написать настоящие игры — «Прыг-скок!» (клон знаменитой игры Pong) и бродилку «Человечек ищет выход», где нужно прыгать по платформам, чтобы добраться до выхода.

Автор книги Джейсон Бриггс занимается программированием с восьми лет. Сегодня он в качестве разработчика и системного архитектора создает программное обеспечение. Кроме того, он пишет статьи о программировании для нескольких профессиональных журналов. «Python для детей» — его первая книга.



Отличная книга с хорошей структурой — от самых простых понятий переходим к сложным. Чтение легкое, много интуитивно понятных примеров, прекрасные иллюстрации и интересные игры для разработки. И, ко всему прочему, самоучитель нашпигован практикой, дающей возможность сразу увидеть результат. Хорошая книга для погружения в программирование.

Максим Тарасов,
старший программист, QIWI



Детские книги на сайте
mann-ivanov-ferber.ru





